TUGAS AKHIR - KI141502

STUDI KINERJA WAVELET TREE PADA VARIASI PERMASALAHAN RANGE QUERY

FENDY
NRP 5113 100 017

Dosen Pembimbing 1
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing 2
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

Halaman ini sengaja dikosongkan

TUGAS AKHIR - KI141502

STUDI KINERJA WAVELET TREE PADA VARIASI PERMASALAHAN RANGE QUERY

FENDY
NRP 5113 100 017

Dosen Pembimbing 1
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing 2
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

Halaman ini sengaja dikosongkan

UNDERGRADUATE THESES - KI141502

WAVELET TREE PERFORMANCE STUDY ON VARIOUS RANGE QUERY PROBLEMS

FENDY
NRP 5113 100 017

Supervisor 1
Arya Yudhi Wijaya, S.Kom., M.Kom.

Supervisor 2
Rully Soelaiman, S.Kom., M.Kom.

INFORMATICS DEPARTMENT
Faculty of Information Technology
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

Halaman ini sengaja dikosongkan

STUDI KINERJA WAVELET TREE PADA VARIASI PERMASALAHAN RANGE QUERY

TUGAS AKHIR

Diajukan Guna Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
Fendy
NRP: 5113 100 017

Disetujui oleh Dosen Pembimbing Tugas Akhir:

Arya Yudhi Wijaya, S.Kom., M.Kom. .....
NIP: 198409042010121002 (Pembimbing 1)

Rully Soelaiman, S.Kom., M.Kom. .....
NIP: 197002131994021001 (Pembimbing 2)

SURABAYA
JANUARI 2017

Halaman ini sengaja dikosongkan

STUDI KINERJA WAVELET TREE PADA VARIASI PERMASALAHAN RANGE QUERY

Nama	: FENDY
NRP	: 5113 100 017
Jurusan	: Teknik Informatika FTIF - ITS
Pembimbing I	: Arya Yudhi Wijaya, S.Kom., M.Kom.
Pembimbing II	: Rully Soelaiman, S.Kom., M.Kom.

Abstrak

Komputasi range query merupakan sebuah masalah yang melibatkan sebuah rentang pencarian. Tipe query secara umum dibagi menjadi dua yaitu, operasi pencarian dan perubahan. Operasi perubahan pada suatu rentang akan menyebabkan perubahan hasil pencarian selanjutnya. Dalam permasalahan range query ini cukup banyak operasi yang harus dilakukan sehingga dibutuhkan struktur data yang mampu mendukung operasi-operasi tersebut secara efisien.

Pada Tugas Akhir ini akan dirancang penyelesaian permasalahan variasi range query antara lain, operasi pencarian bilangan terkecil ke-k, menghitung jumlah elemen yang aktif, mengubah status dari sebuah elemen, dan menukar elemen yang bersebelahan. Struktur data klasik yang sering digunakan untuk permasalahan ini adalah balanced binary search tree atau segmented tree. Namun pada Tugas Akhir ini digunakan Wavelet Tree untuk menyelesaikan operasi-operasi tersebut.

Hasil dari Tugas Akhir ini telah berhasil menyelesaikan permasalahan di atas dengan cukup efisien dengan kompleksitas waktu $O(\lg N)$ per query.

Kata Kunci: range query, wavelet tree, rank query, quantile query

Halaman ini sengaja dikosongkan

WAVELET TREE PERFORMANCE STUDY ON VARIOUS RANGE QUERY PROBLEMS

Name	: FENDY
NRP	: 5113 100 017
Major	: Informatics Department Faculty of IT - ITS
Supervisor I	: Arya Yudhi Wijaya, S.Kom., M.Kom.
Supervisor II	: Rully Soelaiman, S.Kom., M.Kom.

Abstract

Range query computation is a problem which involves a specific range of data. Generally it is divided by two type of query, range search and update. An update operation would have an impact for the following range search query. Typically these kind of problems will have quite large number of queries. Thus, an efficient data structure is needed to support the operations.

This undergraduate thesis will design the problem solving for range query variation such as, find the k-th smallest element, count the number of active elements, toggle an element status, and swap contiguous element. Well known data structures, e.g. balanced binary search tree and segmented tree, are commonly used for solving this problem. Yet, in this undergraduate thesis Wavelet Tree will be used instead for solving those range query variations.

The result shows that wavelet tree has been successfully solve the problem efficiently with overall complexity of $\mathcal{O}(\lg N)$ *each query.*

Keywords: range query, wavelet tree, rank query, quantile query

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa atas pimpinan, penyertaan, dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul :

STUDI KINERJA *WAVELET TREE* PADA VARIASI PERMASALAHAN *RANGE QUERY*.

Penelitian Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.

Dengan selesainya Tugas Akhir ini diharapkan apa yang telah dikerjakan penulis dapat memberikan manfaat bagi perkembangan ilmu pengetahuan terutama di bidang teknologi informasi serta bagi diri penulis sendiri selaku peneliti.

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan Tugas Akhir maupun selama menempuh masa studi antara lain:

- Ayah, Ibu dan keluarga penulis yang selalu memberikan perhatian, dorongan dan kasih sayang yang menjadi semangat utama bagi diri penulis baik selama penulis menempuh masa kuliah maupun pengerjaan Tugas Akhir ini.
- Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah banyak meluangkan waktu untuk memberikan ilmu, nasihat, motivasi, pandangan dan bimbingan kepada penulis baik selama penulis menempuh masa kuliah maupun selama pengerjaan Tugas Akhir ini.
- Bapak Arya Yudhi Wijaya, S.Kom., M.Kom. selaku dosen

pembimbing yang telah memberikan ilmu, dan masukan kepada penulis.

- Seluruh tenaga pengajar dan karyawan Jurusan Teknik Informatika ITS yang telah memberikan ilmu dan waktunya demi berlangsungnya kegiatan belajar mengajar di Jurusan Teknik Informatika ITS.
- Seluruh teman penulis di Jurusan Teknik Informatika ITS yang telah memberikan dukungan dan semangat kepada penulis selama penulis menyelesaikan Tugas Akhir ini.
- Seluruh pihak yang tidak bisa penulis sebutkan satu-persatu yang telah memberikan dukungan selama penulis menyelesaikan Tugas Akhir.

Penulis mohon maaf apabila masih ada kekurangan pada Tugas Akhir ini. Penulis juga mengharapkan kritik dan saran yang membangun untuk pembelajaran dan perbaikan di kemudian hari. Semoga melalui Tugas Akhir ini penulis dapat memberikan kontribusi dan manfaat yang sebaik-baiknya.

Surabaya, Januari 2017

Fendy

DAFTAR ISI

DAFTAR TABEL

Halaman ini sengaja dikosongkan

DAFTAR GAMBAR

Halaman ini sengaja dikosongkan

DAFTAR KODE SUMBER

Halaman ini sengaja dikosongkan

BAB 1
PENDAHULUAN

Pada bab ini akan dijelaskan latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.

1.1 Latar Belakang

Memasuki abad ke-20 perkembangan teknologi informasi semakin cepat. Tidak dapat dipungkiri bahwa hampir semua aspek kehidupan mulai terkomputerisasi. Tujuannya adalah konsistensi dan akurasi. Dengan desain dan implementasi yang tepat sebuah program komputer dapat membantu manusia untuk melakukan sebuah proses yang jika dikerjakan oleh manusia secara langsung akan membutuhkan waktu yang lama dan rentan terhadap *error*. Keunggulan inilah yang menyebabkan semakin banyak penggunaan teknologi informasi pada aspek-aspek kehidupan manusia.

Kesadaran akan keunggulan komputer dalam melakukan suatu pekerjaan menyebabkan semakin besar pula keinginan manusia untuk menyelesaikan masalah-masalah yang sebelumnya tidak dapat diselesaikan tanpa bantuan komputer. Masalah-masalah ini seringkali berkaitan dengan data yang memiliki ukuran sangat besar. Dari data tersebut tentu terdapat informasi-informasi tertentu yang ingin diambil. Dengan metode yang tepat informasi yang diperlukan dapat diambil dengan waktu yang relatif singkat.

Besarnya ukuran data tentu menjadi sebuah masalah tersendiri bagi para ilmuwan. Perkembangan perangkat keras yang ada belum dapat mengimbangi dengan kebutuhan komputasi yang semakin besar. Untuk inilah penyelesaian sebuah masalah membutuhkan pendekat-

an yang tepat, sehingga dapat ditemukan algoritma dan struktur data yang sesuai dengan permasalahan yang ada.

Di era modern ini sudah banyak algoritma dan struktur data yang ditemukan oleh ilmuwan-ilmuwan ternama dunia. Masing-masing algoritma sering kali membutuhkan struktur data yang tepat yang dapat membantu kinerja algoritma tersebut. Permasalahan klasik yang cukup menarik pada topik struktur data adalah *range query* atau model pertanyaan yang melibatkan sebuah rentang tertentu. Penulis tertarik untuk melakukan penelitian untuk membandingkan performa beberapa struktur data yang memiliki kemampuan untuk menangani berbagai variasi dari permasalahan ini.

Topik Tugas Akhir ini mengacu pada sebuah permasalahan yang terdapat pada *Online Judge* SPOJ dengan kode ILKQUERY, ILKQUERY2, dan ILKQUERYIII. Pada permasalahan ini terdapat sebuah kumpulan data berupa angka-angka yang diberikan di awal. Dari kumpulan angka ini akan dilakukan sejumlah pertanyaan dan perubahan atau *update* pada data awal yang diberikan. Struktur data yang diduga dapat membantu penyelesaian permasalahan ini adalah *Wavelet Tree*. Selain dapat menjawab pertanyaan dengan benar, waktu juga menjadi salah satu faktor penting untuk memberikan gambaran tentang performa dari struktur data ini.

Hasil dari Tugas Akhir ini diharapakan dapat memberikan gambaran mengenai performa dari struktur data *Wavelet tree* untuk menyelesaikan permasalahan di atas dan diharapkan dapat memberikan kontribusi pada perkembangan ilmu pengetahuan dan teknologi informasi.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam Tugas Akhir ini adalah sebagai berikut:

1. Bagaimana menganalisis dan menentukan algoritma struktur

data yang tepat untuk menjawab variasi dari permasalahan *range query*?
2. Bagaimana implementasi dari struktur data *Wavelet Tree* yang telah dirancang dalam penyelesaian permasalahan *range query*?

1.3 Batasan Masalah

Permasalahan yang dibahas pada Tugas Akhir ini memiliki beberapa batasan, yaitu sebagai berikut:

1. Batas jumlah maksimum data awal yang diberikan adalah 100.000 angka untuk permasalahan ILKQUERY dan ILKQUERY2. Untuk ILKQUERYIII jumlah maksimum data awal mencapai 1.000.000.
2. Batas maksimum jumlah *query* dan/atau *update* adalah 100.000 perintah.
3. Dataset yang digunakan adalah dataset pada problem SPOJ I LOVE Kd-TREES (ILKQUERY, ILKQUERY2, ILKQUERYIII).

1.4 Tujuan

Tujuan dari Tugas Akhir ini adalah sebagai berikut:

1. Melakukan analisis dan mendesain algoritma dan struktur data untuk menyelesaikan permasalahan variasi *range query* dengan pembuktian (proofing) yang jelas.
2. Membandingkan performa waktu dan memori dari struktur data *Wavelet Tree* untuk menyelesaikan beberapa variasi *range query* yang telah dipaparkan.

1.5 Metodologi

Metodologi yang digunakan dalam pengerjaan Tugas Akhir ini adalah sebagai berikut:

1. Penyusunan proposal Tugas Akhir
   Pada tahap ini dilakukan penyusunan proposal tugas akhir yang berisi permasalahan dan gagasan solusi yang akan diteliti pada permasalahan I Love Kd-Trees, I Love Kd-Trees II, dan I Love Kd-Trees III.
2. Studi literatur
   Pada tahap ini dilakukan pencarian informasi dan studi literatur mengenai pengetahuan atau metode yang dapat digunakan dalam penyelesaian masalah. Informasi didapatkan dari materi-materi yang berhubungan dengan algoritma dan struktur data yang digunakan untuk penyelesaian permasalahan ini, materi-materi tersebut didapatkan dari buku, jurnal, maupun internet.
3. Desain
   Pada tahap ini dilakukan desain rancangan algoritma dan struktur data yang digunakan dalam solusi untuk pemecahan masalah I Love Kd-Trees, I Love Kd-Trees II, dan I Love Kd-Trees III.
4. Implementasi perangkat lunak
   Pada tahap ini dilakukan implementasi atau realiasi dari rancangan desain algoritma dan struktur data yang telah dibangun pada tahap desain ke dalam bentuk program.
5. Uji coba dan evaluasi
   Pada tahap ini dilakukan uji coba kebenaran implementasi dan uji coba generalisasi. Pengujian kebenaran dilakukan pada sistem penilaian daring SPOJ sesuai dengan masalah yang dikerjakan untuk diuji apakah luaran dari program telah sesuai. Uji coba generalisasi digunakan untuk menguji struktur data generalisasi pada variasi permasalahan lain.

6. Penyusunan buku Tugas Akhir
   Pada tahap ini dilakukan penyusunan buku Tugas Akhir yang berisi dokumentasi hasil pengerjaan Tugas Akhir.

1.6 Sistematika Penulisan

Berikut adalah sistematika penulisan buku Tugas Akhir ini:

1. BABI: PENDAHULUAN
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma penyelesaian yang digunakan dalam Tugas Akhir
3. BAB III: DESAIN
   Bab ini berisi desain algoritma dan struktur data yang digunakan dalam penyelesaian permasalahan.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi implementasi berdasarkan desain algortima dan struktur data yang telah dilakukan pada tahap desain.
5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisi uji coba dan evaluasi dari hasil implementasi yang telah dilakukan pada tahap implementasi.
6. BAB VI: KESIMPULAN
   Bab ini berisi kesimpulan yang didapat dari hasil uji coba yang telah dilakukan.

Halaman ini sengaja dikosongkan

BAB 2

DASAR TEORI

Pada bab ini akan dijelaskan mengenai dasar teori yang menjadi dasar pengerjaan Tugas Akhir ini.

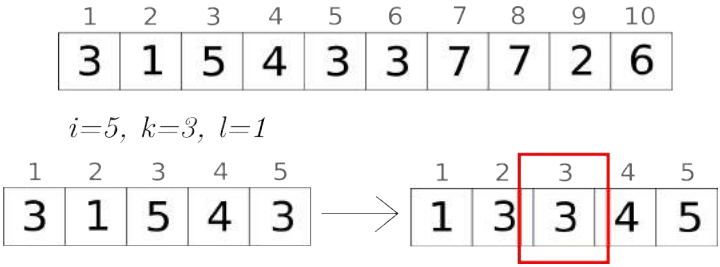
2.1 Deskripsi Permasalahan

Range query merupakan sebuah permasalahan klasik dalam pengembangan sebuah struktur data yang efisien. Pada Tugas Akhir ini diangkat permasalahan mengenai variasi *range query*. Sebuah *range query* memiliki dua faktor yang penting yaitu, data yang bagaimana yang ingin dicari dan pada indeks mana pertanyaan tersebut akan dijawab. Sebagai contoh, *query* yang dikerjakan pada indeks $[1..5]$ akan menghasilkan output yang berbeda dengan *query* yang dikerjakan pada indeks $[3..7]$. Pertanyaan mengenai data bagaimana yang ingin dicari pada sebuah jenis *query* juga sangat mempengaruhi struktur data apa yang dapat menangani *query* tersebut dengan efisien. Efisien yang dimaksud adalah efisien baik dari segi *runtime* maupun dari segi *memory* yang dibutuhkan untuk membangun struktur data tersebut.

Mula-mula terdapat N buah bilangan sebagai data awal. Kemudian terdapat serangkaian operasi yang akan dilakukan pada data awal yang telah diberikan. Suatu operasi dapat berupa sebuah *query* atau sebuah tindakan yang mengubah kondisi data. Variasi dari operasi-operasi yang menjadi permasalahan dalam Tugas Akhir ini akan dijelaskan pada subbab-subbab di bawah.

2.1.1 Operasi Mencari Bilangan Terkecil ke-K

Terdapat 3 parameter pada operasi ini yaitu i, l, dan k. Dari tiga parameter tersebut diajukan sebuah pertanyaan yaitu, misalkan sebuah bilangan d merupakan bilangan terkecil ke-k dari indeks 1 hingga indeks ke-i maka pada indeks ke berapakah bilangan d muncul yang ke-l kali. Jika jumlah kemunculan bilangan d kurang dari l maka jawaban dari pertanyaan ini adalah -1.



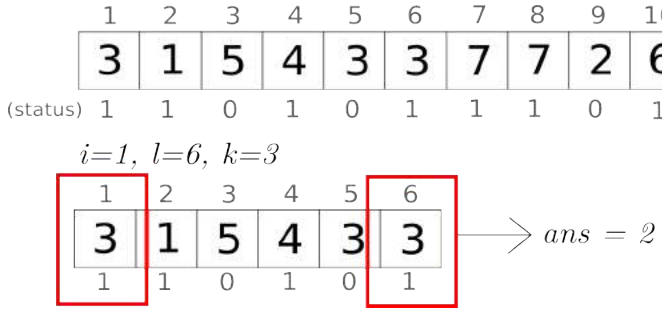
Gambar 2.1: Ilustrasi operasi mencari bilangan terkecil ke-K.

2.1.2 Operasi Menghitung Jumlah Elemen yang Aktif

Sama dengan operasi mencari bilangan terkecil ke-K operasi ini juga memiliki 3 parameter yaitu, i, l, dan k. Namun masing-masing memiliki representasi yang berbeda. Parameter i dan l menyatakan *range* dari pertanyaan ini dengan parameter i sebagai batas kiri serta parameter l sebagai batas kanan. Parameter k menyatakan bilangan yang ingin dicari. Sehingga pertanyaan yang harus dijawab pada operasi ini adalah berapa bilangan k yang memiliki status aktif pada *range* $[i..l]$.

2.1.3 Operasi Mengubah Status dari Sebuah Elemen

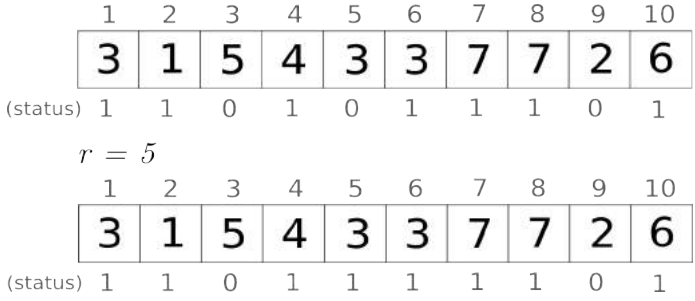
Pada operasi ini status dari sebuah elemen pada array akan diubah menjadi lawan dari status elemen tersebut sekarang. Jika status elemen tersebut adalah aktif maka status dari elemen ini akan diubah



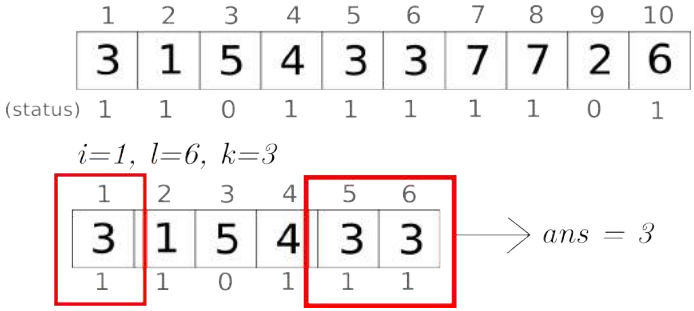
Gambar 2.2: Ilustrasi operasi menghitung jumlah elemen yang aktif.

menjadi tidak aktif dan sebaliknya jika status dari elemen tersebut adalah tidak aktif maka status dari elemen ini akan diubah menjadi aktif. Operasi ini sangat berkaitan dengan Operasi Menghitung Jumlah Elemen yang Aktif. Parameter yang diberikan untuk operasi ini adalah r yang menyatakan indeks elemen yang akan diubah statusnya.

Pada Gambar 2.4 diperlihatkan bagaimana operasi ini dapat menyebabkan perubahan hasil pada operasi 2.1.2 walaupun parameter yang diberikan pada masing-masing operasi sama persis.



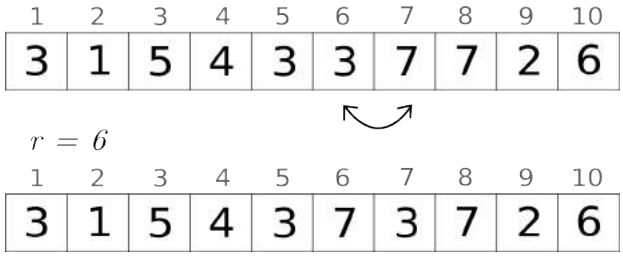
Gambar 2.3: Ilustrasi operasi mengubah status dari sebuah elemen.



Gambar 2.4: Ilustrasi operasi menghitung jumlah elemen yang aktif setelah terjadi perubahan status dari sebuah elemen.

2.1.4 Operasi Menukar Elemen yang Bersebelahan

Operasi ini akan menukar sebuah elemen di indeks r dengan elemen di sebelah kanannya atau dengan kata lain elemen yang memiliki indeks $r + 1$.



Gambar 2.5: Ilustrasi operasi menukar elemen yang bersebelahan.

2.2 Dataset yang Digunakan

Variasi permasalahan *range query* yang akan dibahas pada Tugas Akhir ini akan menggunakan dataset yang diambil dari situs penilaian adaring SPOJ dengan judul,

1. *I Love Kd-Trees*

Operasi yang diperbolehkan pada operasi ini adalah,

(a) Operasi Mencari Bilangan Terkecil ke-K.

Dengan batasan masukan yang diperbolehkan pada permasalahan ini adalah,

- $1 \leq N \leq 10^5$
- $1 \leq Q \leq 10^5$
- $-10^9 \leq a_i \leq 10^9$
- $0 < k < i < N$
- $1 \leq l \leq N$

Dimana,

- N merupakan jumlah data yang terdapat pada masukan.
- Q merupakan jumlah operasi yang harus dikerjakan.
- a_i merupakan data pada indeks ke-i.
- i, l, k merupakan parameter untuk Operasi Mencari Bilangan Terkecil ke-K.

2. *I Love Kd-Trees II*

Operasi yang diperbolehkan pada operasi ini adalah,

(a) Operasi Menghitung Jumlah Elemen yang Aktif.

(b) Operasi Mengubah Status dari Sebuah Elemen.

Dengan batasan masukan yang diperbolehkan pada permasalahan ini adalah,

- $1 \leq N \leq 10^5$
- $1 \leq Q \leq 10^5$
- $-10^9 \leq a_i \leq 10^9$
- $0 < q < 2$
- $0 \leq i \leq l < N$
- $-10^9 \leq k \leq 10^9$
- $0 \leq r < N$

Dimana,

- N merupakan jumlah data yang terdapat pada masukan.
- Q merupakan jumlah operasi yang harus dikerjakan.
- a_i merupakan data pada indeks ke-i.
- q merupakan jenis operasi yang harus dikerjakan.

- i, l, k merupakan parameter untuk Operasi Menghitung Jumlah Elemen yang Aktif.
- r merupakan parameter untuk Operasi Mengubah Status dari Sebuah Elemen.

3. *I Love Kd-Trees III*

   Operasi yang diperbolehkan pada operasi ini adalah,

   (a) Operasi Mencari Bilangan Terkecil ke-K
   (b) Operasi Menukar Elemen yang Bersebelahan

   Dengan batasan masukan yang diperbolehkan pada permasalahan ini adalah,

   - $1 \leq N \leq 10^6$
   - $1 \leq Q \leq 10^5$
   - $-10^9 \leq a_i \leq 10^9$
   - $0 < q < 2$
   - $0 \leq i < N$
   - $1 \leq l \leq N$
   - $1 \leq k \leq i + 1$
   - $0 \leq r < N$

   Dimana,

   - N merupakan jumlah data yang terdapat pada masukan.
   - Q merupakan jumlah operasi yang harus dikerjakan.
   - a_i merupakan data pada indeks ke-i.
   - q merupakan jenis operasi yang harus dikerjakan.
   - i, l, k merupakan parameter untuk Operasi Menghitung Jumlah Elemen yang Aktif.
   - r merupakan parameter untuk Operasi Mengubah Status dari Sebuah Elemen.

2.3 *Wavelet* Trees

Wavelet tree diciptakan pertama kali sebagai sebuah struktur data yang merepresentasikan sebuah sekuens untuk menjawab beberapa jenis *query* [1]. Dalam perkembangannya *wavelet tree* telah dipandang sebagai sebuah struktur yang mampu menyelesaikan variasi

permasalahan mulai dari *text-indexing* hingga ke persoalan *computational geometry* [3]. Secara umum, *wavelet tree* dibangun pada sebuah sekuens S yang terdiri dari himpunan simbol Σ.

Struktur dasar dari *wavelet tree* merupakan sebuah compelete binary *tree*. Setiap *node* pada *wavelet tree* mewakili sebuah rentang tertentu dari sekuens asal. Uniknya rentang ini selalu konsekutif namun selalu terurut atau yang biasa disebut dengan subsekuens. Simbol-simbol yang ada pada suatu *node* juga merupakan himpunan bagian dari himpunan semesta untuk *wavelet tree* ini.

```
Build(part)
1  if V is leaf
2       Σ_L ⊂ Σ_V
3       Σ_R ⊂ Σ_V
4       LeftPartition = {S_i | S_i ∈ Σ_L}
5       RightPartition = {S_i | S_i > Σ_R}
6       Build(LeftPartition)
7       Build(RightPartition)
```

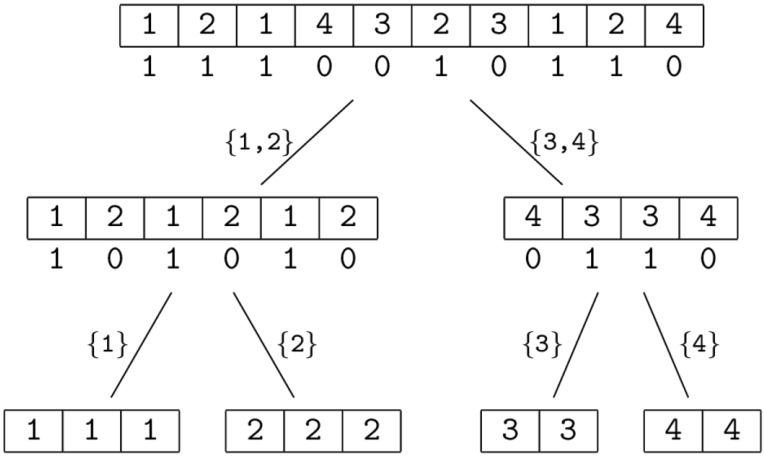
Gambar 2.6: Pseudocode Build

Setiap *node* pada *wavelet tree* akan memiliki anak kiri dan kanan jika dan hanya jika pada *node* tersebut masih merepresentasikan > 1 simbol. Himpunan simbol tersebut kemudian dibagi menjadi dua sama besar. Semua simbol yang termasuk dalam himpunan bagian pertama akan dipartisi menjadi anak kiri dan sebaliknya semua simbol yang termasuk dalam himpunan bagian kedua akan dipartisi menjadi anak kanan.

Proses ini dilakukan secara rekursif untuk setiap subtree yang terbentuk dan berhenti bila mencapai *node leaf*. Melalui properti ini

secara implisit definisi dari *node leaf* telah dirumuskan yaitu *node* yang hanya merepresentasikan sebuah simbol.

Σ_L merupakan himpunan bagian dari Σ_V yang berisi simbol-simbol oada anak kiri begitu juga dengan Σ_R yang akan berisi simbol-simbol pada anak kanan. Kedua himpunan ini saling *mutually exclusive*. Dimana sebuah elemen pada Σ_V hanya dapat tergabung pada salah satu himpunan bagian. Kemudian fungsi Build dipanggil pada sekuens awal S yang akan menghasilkan struktur *wavelet tree* seperti pada Gambar 2.7.



Gambar 2.7: Struktur *wavelet tree* setelah proses build dengan sekuens awal $S = \{1, 2, 1, 4, 3, 2, 3, 1, 2, 4\}$. Pada ilustrasi ini ditampilkan elemen pada masing-masing *node* untuk memberikan gambaran umum namun data tersebut tidak disimpan pada struktur *wavelet tree*.

Teorema 2.1

Jika T adalah sebuah *wavelet tree* dari sekuens S dengan himpunan simbol Σ maka *wavelet tree* T adalah sebuah complete binary *tree* dengan tinggi $H(T) = \mathcal{O}(\lg \sigma)$ dimana $\sigma = |\Sigma|$.

Proof Misalkan $H(\sigma)$ adalah tinggi dari sebuah *wavelet tree* dengan total σ simbol. Maka,

$$H(\sigma) = \begin{cases} H(\frac{\sigma}{2}) + 1, & \text{if } \sigma > 1. \\ 1, & \text{if } \sigma = 1. \end{cases}$$

Dengan master theorm maka rekurens ini terselesaikan menjadi $H(\lg \sigma)$.

2.3.1 Bit vector

Pada Gambar 2.7 tampak terdapat deret bilangan 0 dan 1 dibawah elemen masing-masing *node*. Bilangan tersebut didefinisikan sebagai bitvector. Misalkan B adalah sebuah bitvector maka,

$$B_i = \begin{cases} B_i = 1, & \text{if } S_i \in \Sigma_L. \\ B_i = 0, & \text{if } S_i \in \Sigma_R. \end{cases}$$

Informasi ini merupakan informasi dasar yang diperlukan untuk melakukan penulusuran pada *wavelet tree*. Berawal dari informasi ini dapat dibentuk informasi lain yang dapat membantu operasi-operasi yang ada pada *wavelet tree*. Meskipun nantinya informasi ini tidak digunakan namun informasi ini akan terbukti esensial pada struktur data ini.

2.3.2 $MapLeft$ dan $MapRight$

Operasi $MapLeft(i)$ didefinisikan sebagai banyak elemen hingga indeks i atau pada rentang $[0..i]$ yang memiliki nilai $B_x = 1 | x \leq i$. Melalui bitvector yang telah terbentuk dapat dilakukan iterasi dari indeks 0 hingga i untuk mencari banyak bitvector yang memiliki nilai $= 1$. Namun solusi ini mempunyai kompleksitas sebesar $\mathcal{O}(n)$ dengan n adalah panjang sekuens, tidak cukup efisien.

Sebuah solusi sederhana adalah dengan membangun sebuah array yang merupakan prefix sum dari bitvector. Misalkan P adalah array prefix sum dari bitvector B maka,

$$P_i = \begin{cases} P_i = B_i + P_{i-1}, & \text{if } i > 0. \\ P_i = B_i, & \text{otherwise.} \end{cases}$$

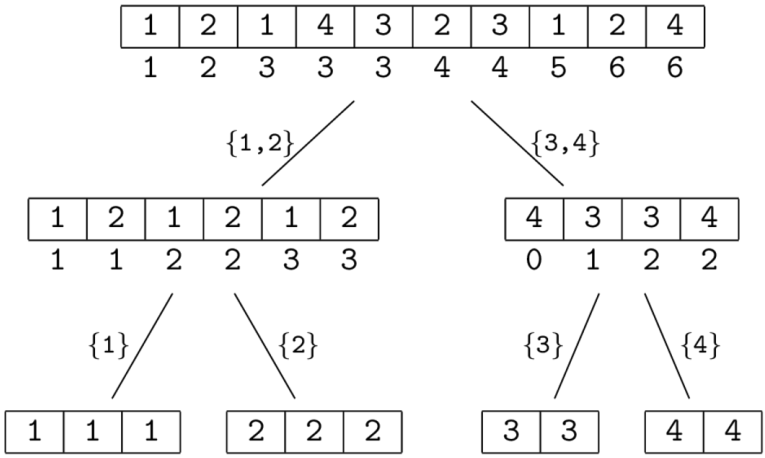
Dengan solusi ini maka $MapLeft(i)$ dapat diselesaikan dengan kompleksitas $\mathcal{O}(1)$ dengan tambahan memori sebesar $\mathcal{O}(n)$ per level dari *wavelet tree*. Sehingga memori total yang dibutuhkan adalah $\mathcal{O}(n \lg \sigma)$.

Untuk menyelesaikan operasi $MapRight(i)$ dapat dilakukan hal yang sama dengan membangun sebuah array prefix sum namun dengan menjumlahkan bit-bit yang bernilai 0. Menarik diperhatikan bahwa sebenarnya hal ini tidak perlu dilakukan karena operasi $MapRight$ ini dapat diperoleh dari operasi $MapLeft$ dengan persamaan,

$$MapRight(i) = (i + 1) - MapLeft(i)$$

Seperti yang telah dijelaskan bahwa operasi $MapLeft$ mengembalikan nilai luaran berupa banyak elemen hingga indeks i dengan bitvector bernilai 1. Pada sembarang elemen dalam sekuens *wavelet tree* bitvector untuk elemen tersebut hanya mungkin bernilai 1 atau 0.

Dengan ini dapat dipastikan bahwa jika terdapat $i + 1$ elemen dan sebanyak $MapLeft(i)$ elemen memiliki bitvector $= 1$ maka terdapat sebanyak $(i+1) - MapLeft(i)$ elemen yang memiliki bitvector $= 0$. Penambahan satu elemen dikarenakan penulis menggunakan indeks yang dimulai dari 0. Pada Gambar 2.8 ditunjukkan struktur dari *wavelet tree* yang menggunakan array prefix sum..



Gambar 2.8: Struktur *wavelet tree* dengan sekuens awal $S = \{1, 2, 1, 4, 3, 2, 3, 1, 2, 4\}$. Dibawah setiap elemen merupakan array prefix sum yang merepresentasikan nilai luaran dari $MapLeft$ untuk semua indeks.

Lebih lanjut operasi ini dapat digunakan untuk mengetahui rentang pencarian saat melakukan traversal ke anak kiri atau kanan dari sebuah *node* V. Operasi $MapLeft(i)$ dapat dilihat sebagai banyak elemen yang dipetakan pada anak kiri hingga indeks ke-i.

Jika pencarian dilanjutkan ke anak kiri dari V maka rentang pencarian berubah dari $[0..i]$ menjadi $[0..MapLeft(i) - 1]$ atau sampai elemen ke-$MapLeft(i)$ pada anak kiri. Sebaliknya jika pencarian dilanjutkan pada anak kanan dari V maka rentang pencarian menjadi $[0..MapRight(i) - 1]$.

2.3.3 *Rank Query*

Didefinisikan $rank(i, k)$ adalah operasi yang mengembalikan nilai luaran berupa banyak simbol k pada rentang $[0..1]$. Untuk menye-

```
Rank(V, i, k)
1  if V is leaf
2      return i + 1
3  elseif k ∈ Σ_L
4      Rank(V.left, MapLeft(i) − 1, k)
5  else
6      Rank(V.right, MapRight(i) − 1, k)
```

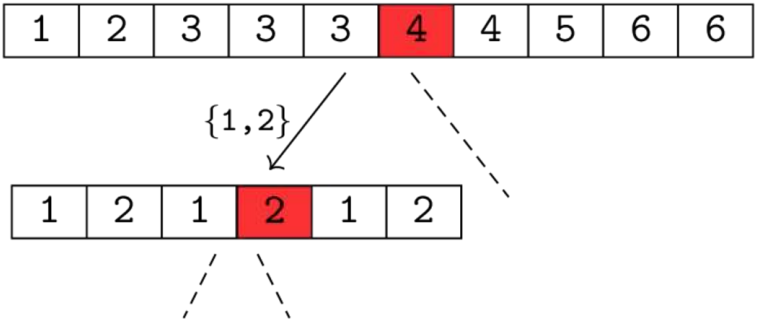
Gambar 2.9: Pseudocode *Rank Query*.

lesaikan operasi ini dapat dilakukan strategi yang ditunjukkan pada Gambar 2.9.

Sebagai contoh akan dilakukan proses pencarian untuk $rank(5, 2)$. Simbol yang akan dicari adalah simbol 2. Simbol ini dipetakan ke anak kiri sehingga pencarian akan dilanjutkan ke anak kiri. Pada Gambar 2.10 ditunjukkan bagian-bagian yang terlibat pada tahap ini. Seperti telah dipaparkan pada Subbab 2.3.2 bahwa ketika proses traversal pada *wavelet tree* dilanjutkan ke anak kiri atau kanan maka rentang pencarian akan mengalami perubahan.

Secara intuitif hal ini perlu dilakukan dikarenakan pada rentang pencarian awal $[0..i]$ terdapat elemen-elemen yang dipetakan ke anak kiri serta ada juga yang dipetakan ke anak kanan. Untuk contoh kasus ini pencarian akan dilanjutkan ke anak kiri dan rentang pencarian berubah menjadi $[0..3]$.

Pencarian dilanjutkan ke anak kanan karena $2 \in \Sigma_R$. Rentang pencarian menjadi $[0..1]$ karena $MapRight(i) = 2$. Sehingga nilai dari $rank(5, 2) = 2$. Perlu diingat bahwa *node leaf* hanya merepresentasikan sebuah simbol yang dengan kata lain i menyatakan jumlah kemunculan simbol tersebut muncul hingga indeks i. Sehingga jawaban dari operasi ini adalah $i + 1$, penambahan satu merupakan



Gambar 2.10: Operasi $rank(5, 2)$ dimulai pada *root*. Bagian yang diarsir warna merah menunjukkan nilai dari $MapLeft(i)$.

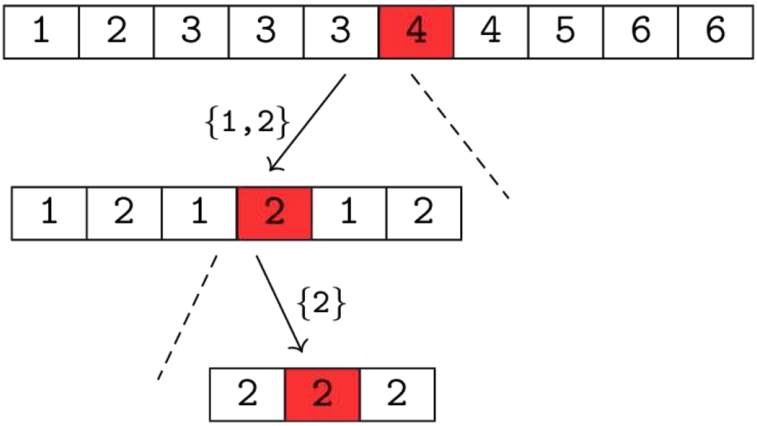
akibat penggunan 0-based indeks. Pada Gambar 2.11 ditunjukkan proses pencarian dari awal hingga akhir.

Modifikasi sederhana dari operasi ini adalah perubahan dari indeks pencarian menjadi $[i..j]$. Modifikasi ini tidak mengubah proses dari operasi $rank$ itu sendiri dikarenakan modifikasi ini dapat diselesaikan dengan $rank(j) - rank(i-1)$. Hal ini dapat dilakukan karena berlakunya prinsip inklusi-eksklusi.

Dari algoritma yang telah dipaparkan terlihat bahwa pada proses traversal untuk operasi $rank$ ini memiliki kompleksitas $\mathcal{O}(\lg \sigma)$. Pada saat mencapai sembarang *node* pada *wavelet tree* T maka proses akan dilanjutkan ke anak kiri atau anak kanan yang menunjukkan bahwa proses traversal hanya akan mengunjungi paling banyak $\lg \sigma$ *node*.

2.3.4 *Quantile Query*

Operasi $quantile_k(i)$ didefinisikan sebagai pencarian simbol ke-k pada rentang $[0..i]$. Strategi untuk menyelesaikan permasalahan ini ditunjukkan pada Gambar 2.12.



Gambar 2.11: Setelah dicapai *node leaf* maka jawaban untuk operasi $rank(5, 2)$ adalah 2, operasi berhenti ketika pemanggilan $rank(5, 1)$. Selain *node leaf* struktur yang ditampilkan merupakan representasi dari array $MapLeft$.

Sebagai contoh operasi $quantile_4(4)$ akan mengembalikan simbol 3 dimana elemen-elemen hingga indeks 4 adalah $1, 2, 1, 4, 3$ jika ditulis terurut menjadi $1, 1, 2, 3, 4$ maka simbol ke-4 adalah 3.

Mula-mula proses diawali pada *node root* dari *wavelet tree* kemudian dilakukan perbandingan antara nilai k yang dicari dengan $MapLeft(i)$. Maksud dari tahapan ini adalah mengetahui simbol yang dicari berada pada subtree kiri atau subtree kanan dengan membandingkan nilai k dengan $MapLeft(i)$.

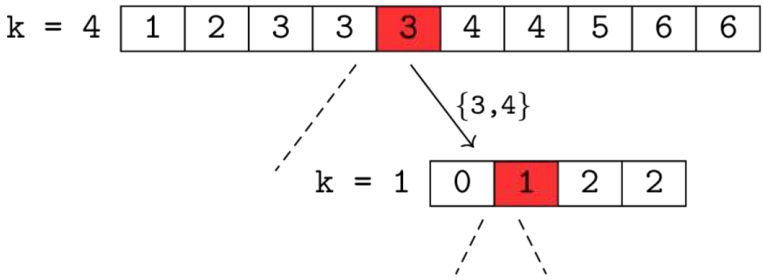
Jika nilai $k \leq MapLeft(i)$ maka diketahui terdapat cukup elemen pada subtree kiri untuk mencari elemen ke-k pada rentang yang diberikan. Sebaliknya jika $k > MapLeft(i)$ menandakan bahwa tidak terdapat cukup elemen untuk menjawab pertanyaan ini pada subtree kiri oleh karena itu pencarian akan dilanjutkan ke subtree kanan.

```
Quantile(V, i, k)
1  if V is leaf
2      return V
3  elseif k ≤ MapLeft(i)
4      Quantile(V.left, MapLeft(i) − 1, k)
5  else
6      Quantile(V.right, MapRight(i) − 1, k − MapLeft(i))
```

Gambar 2.12: Pseudocode *Quantile Query*.

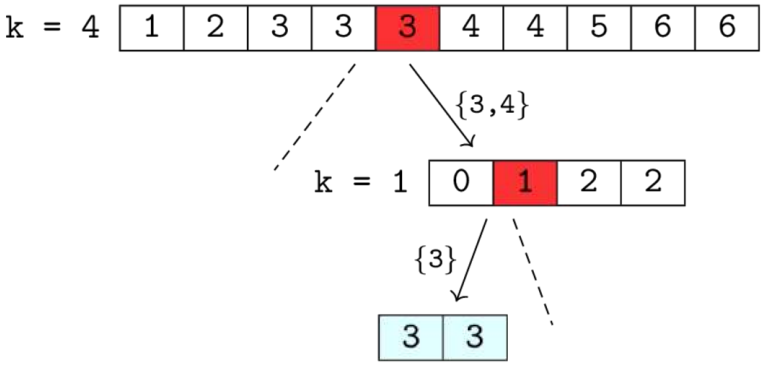
Jika traversal dilanjutkan ke subtree kanan maka pencarian tidak lagi mencari elemen ke-k melainkan menjadi elemen ke-$(k - MapLeft(i))$. Hal ini dikarenakan sudah terdapat sejumlah $MapLeft(i)$ elemen yang telah masuk pada subtree kiri dan tidak mungkin menjadi jawaban dari operasi ini.

Pada contoh $k = 4$ dan $MapLeft(4) = 3$ maka pencarian dilanjutkan ke anak kanan dari *root*. Ilustrasi pencarian ini ditunjukkan pada Gambar 2.13.



Gambar 2.13: $quantile_4(4)$ dimulai pada *node root* kemudian dilanjutkan ke anak kanan. Ditunjukkan bahwa nilai k juga mengalami perubahan saat melakukan traversal ke subtree kanan.

Selanjutnya secara rekursif dilanjutkan proses dari operasi $quantile$ ke anak kanan. Nilai dari k sekarang adalah $4 - 3 = 1$ serta nilai dari $i = MapRight(4) - 1 = 1$. Kondisi $k \leq MapLeft(i)$ terpenuhi sehingga pencarian dilanjutkan pada anak kiri yang merupakan *node leaf*. Sehingga jawaban dari $quantile_4(4)$ adalah 3. Ilustrasi lengkap untuk operasi $quantile_4(4)$ ditunjukkan pada Gambar 2.14.



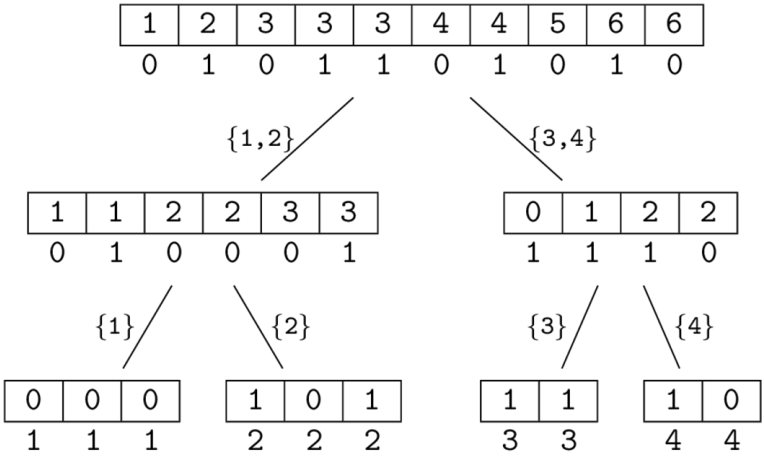
Gambar 2.14: Proses penulusuran untuk operasi $quantile_4(4)$

Kompleksitas dari operasi $quantile$ adalah $\mathcal{O}(\lg \sigma)$ yang merupakan tinggi dari *wavelet tree* itu sendiri. Dari proses pencarian yang telah dipaparkan di atas secara intuitif jelas bahwa proses traversal hanya dapat bergerak ke anak kiri atau kanan. Sehingga banyak *node* yang dikunjungi dalam satu operasi adalah $\mathcal{O}(\lg \sigma)$.

2.3.5 Toggle *Update*

Operasi ini membutuhkan informasi tambahan yang didefinisikan terlebih dahulu sebelumnya yaitu, status dari setiap elemen. Operasi $toggle(i)$ melakukan perubahan status elemen pada posisi ke-i. Status elemen akan diubah menjadi aktif jika status elemen tersebut

sekarang adalah non-aktif dan sebaliknya.



Gambar 2.15: Struktur dari *wavelet tree* setelah modifikasi untuk mendukung operasi $toggle$ dengan $A = \{0, 1, 0, 1, 1, 0, 1, 0, 1, 0\}$. Dimana A adalah status dari masing-masing elemen. Array dengan bingkai menyatakan data yang disimpan pada setiap *node*.

R. Castro, et. al telah menujukkan bahwa operasi ini dapat memberikan pengaruh pada operasi $rank$ dan $quantile$ yang telah dibahas sebelumnya [4]. Solusi yang ditawarkan adalah dengan menambahkan sebuah informasi tambahan pada setiap *node* dari *wavelet tree* yaitu, $ActiveLeft$. Informasi ini menyerupai informasi $MapLeft$ namun ditambahkan bitvector ke-i akan bernilai satu jika status elemen tersebut adalah aktif. Dengan penambahan informasi ini operasi $rank$ dan $quantile$ akan memiliki kompleksitas tetap $\mathcal{O}(\lg \sigma)$.

Pada permasalahan yang telah dipaparkan operasi $toggle$ ini akan digabungkan hanya dengan operasi $rank$. Penggabungan kedua operasi ini ternyata tidak membutuhkan informasi tambahan untuk setiap *node*. Informasi tetap perlu ditambahkan pada *node leaf* di-

mana ketika mencapai *node leaf* operasi $rank$ tidak dapat langsung mengembalikan nilai luaran $i + 1$ melainkan perlu melakukan perhitungan elemen yang aktif pada rentang $[0..i]$.

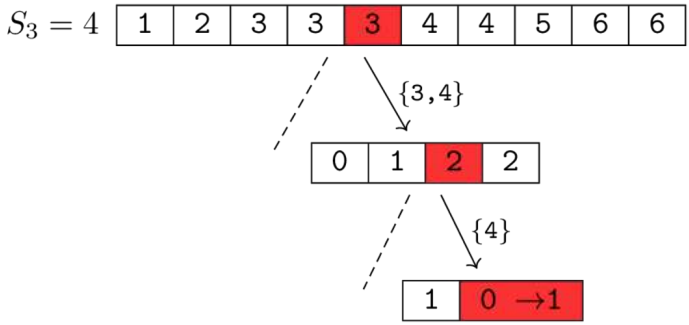
Permasalahan ini dapat dilihat kembali sebagai bitvector yang akan bernilai 1 jika elemen tersebut dalam kondisi aktif dan bernilai 0 jika elemen tersebut dalam kondisi non-aktif. Representasi ini menunjukkan bahwa dapat pula dibangun sebuah array prefix sum bitvecor untuk menyatakan jumlah elemen yang aktif pada rentang $[0..i]$. Namun kelemahan dari prefix sum adalah tidak dapat mendukung operasi *update* dengan cukup efisien. Operasi *update* mengharuskan mengubah semua data dari posisi i hingga posisi terakhir.

```
Toggle(V, i, k)
1  if V is leaf
2      BITupdate(i)
3  elseif k ∈ Σ_L
4      Toggle(V.left, MapLeft(i) − 1, k)
5  else
6      Toggle(V.right, MapRight(i) − 1, k)
```

Gambar 2.16: Pseudocode Toggle *Update*.

Binary Indexed *Tree* atau Fenwick *Tree* merupakan sebuah struktur data yang memiliki karakteristik mirip dengan array prefix sum yang juga mendukung operasi *update*. Kedua operasi ini dapat diselesaikan dengan kompleksitas waktu $\mathcal{O}(\lg n)$ dan kompleksitas memori $\mathcal{O}(n)$ dengan n adalah jumlah data.

Untuk mendukung operasi ini akan ditambahkan struktur data BIT pada setiap *node leaf*. Pada Gambar 2.15 merupakan struktur dari *wavelet tree* yang telah dimodifikasi. Strategi untuk menyelesaikan operasi $toggle$ ditunjukkan pada Gambar 2.16.



Gambar 2.17: Ilustrasi dari proses operasi $toggle(3)$.

Secara garis besar algoritma yang digunakan untuk melakukan perubahan status adalah dengan melakukan traversal pada *wavelet tree*, bertujuan untuk menemukan *node leaf* yang menyimpan simbol pada elemen ke-i. Setelah mencapai *node leaf* maka dilakukan *update* pada struktur BIT. Proses traversal sangat menyerupai proses traversal pada operasi $rank$. Sebagai contoh operasi $toggle(3)$ ditunjukkan pada Gambar 2.17.

2.3.6 Swap *Update*

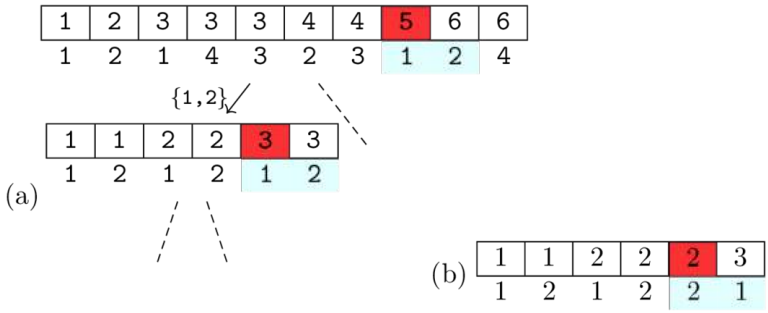
Pada operasi $update(i)$ dilakukan penukaran elemen pada indeks ke-i dan $i+1$. Hal ini dapat memberikan pengaruh yang besar pada *query* yang ditanyakan setelah operasi *update* ini dilakukan. Meskipun demikian perubahan struktur pada *wavelet tree* akibat operasi ini hanya akan berpengaruh pada nilai dari operasi $MapLeft(i)$ jika dan hanya jika simbol pada posisi ke-i dan $i+1$ berbeda satu sama lain.

Jika kedua simbol pada posisi tersebut merupakan simbol yang sama maka perubahan akan tidak akan terjadi pada *node* tersebut. Hal ini dikarenakan kedua simbol tersebut merupakan anggota subtree yang sama sehingga operasi $swap$ dilanjutkan ke subtree kiri

```
Swap(V, i)
1  if S_i ∈ Σ_L and S_{i+1} ∈ Σ_L
2       Swap(V.left, MapLeft(i) − 1)
3  elseif S_i ∈ Σ_R and S_{i+1} ∈ Σ_R
4       Swap(V.left, MapRight(i) − 1)
5  else
6       if S_i ∈ Σ_L
7            MapLeft(i) = MapLeft(i) − 1
8       else
9            MapLeft(i) = MapLeft(i) + 1
```

Gambar 2.18: Pseudocode Swap *Update*.

atau kanan tergantung simbol tersebut masuk ke himpunan yang mana. Pseduocode untuk operasi ini ditunjukkan pada Gambar 2.18.



Gambar 2.19: (a) Ilustrasi dari eksekusi operasi $swap(7)$. (b) Struktur akhir dari *node* yang mengalami perubahan.

Pada Gambar 2.19 ditunjukkan contoh operasi $swap(7)$. Pada *node root* $MapLeft(7) = 5$ dan simbol pada indeks 7 dan 8 masuk pada

himpunan Σ_L yang menyatakan kedua simbol tersebut merupakan anggota dari subtree kiri. Maka nilai $i = MapLeft(i) - 1 = 4$. Kemudian pada anak kiri ternyata simbol pada indeks 4 dan 5 sudah tidak lagi tergabung pada subtree yang sama sehingga pasti terdapat perubahan pada nilai $MapLeft$.

Simbol pada indeks ke-4 merupakan anggota dari subtree kiri sehingga dapat dipastikan bahwa nilai pada indeks ke-5 merupakan anggota dari subtree kanan. Hal ini menyebabkan pada indeks ke-4 simbol tersebut akan tergabung pada subtree kanan yang secara tidak langsung menyatakan bahwa nilai $B_4 = 0$ dan menyebabkan nilai $MapLeft(4) = 2$. Proses dihentikan karena kedua elemen sudah terpisah ke subtree yang berbeda.

Halaman ini sengaja dikosongkan

BAB 3

DESAIN

Pada bab ini akan dijelaskan desain sistem yang digunakan untuk menyelesaikan permasalahan-permasalahan pada Tugas Akhir ini.

3.1 Permasalahan I Love Kd-Trees

Pada permasalahan ini sistem akan dibangun untuk menyelesaikan hanya satu jenis operasi yaitu, Operasi Mencari Bilangan Terkecil ke-K.

3.1.1 Deskripsi Umum Sistem

Kemudian untuk membantu menyelesaikan Operasi Mencari Bilangan Terkecil ke-K maka array global $occurences$ diinisialisasi dengan posisi kemunculan masing-masing elemen terurut dari kiri ke kanan. Penggunaan array ini telah dijabarkan pada 2.1.1.

Selanjutnya Operasi Mencari Bilangan Terkecil ke-K akan menerima 3 bilangan bulat i, l, dan k yang masing-masing menyatakan batas kanan dari rentang pencarian, elemen terkecil ke-k yang harus dicari, dan kemunculan ke berapa dari elemen tersebut. Pseudocode Fungsi Main ditunjukkan pada Gambar 3.1.

```
1   let elements be a new dynamic array
2   let map_left be a dynamic arrays of dynamic arrays
3   let occurences[1..size] be an array of dynamic arrays
4   let symbols be a new dictionary
5   N = Input() // Number of elements
6   Q = Input() // Number of operations
7   for i = 0 to N − 1
8       elements[i] = Input()
9   Init(elements)
10  for i = 0 to N − 1
11      occurences[elements[i]].push(i)
12  Build(elements, 1, 0, size)
13  for q = 0 to Q − 1
14      i = Input()
15      l = Input()
16      k = Input()
17      Print(KthQuery(i, l, k))
```

Gambar 3.1: Pseudocode Fungsi Main I Love Kd-Trees

3.1.2 Desain Fungsi Init

Fungsi Init ini menerima satu parameter array yang berisi bilangan bulat yaitu $elements$. Berdasarkan array ini nantinya akan dibuat sebuah kamus $symbols$ yang menyimpan simbol-simbol apa saja yang muncul pada data awal. Pseudocode Fungsi Init ditunjukkan pada Gambar 3.2.

3.1.3 Desain Fungsi Build

Fungsi Build akan menerima 4 buah parameter yaitu $partition$, $node$, $left$, dan $right$. $partition$ merupakan sebuah array dengan

```
1  sorted = elements.copy()
2  sort(sorted)
3  for element ∈ elements
4      symbols.insert(element)
5  for i to N − 1
6      elements[i] = symbols[elements[i]]
7  map_left.assign(3 ∗ size)
```

Gambar 3.2: Pseudocode Fungsi Init I Love Kd-Trees

ukuran yang dinamis. $node$ merupakan pointer yang menunjuk pada *node* mana struktur pada *wavelet tree* itu sedang dibuat. Sedangkan, $left$ dan $right$ merupakan bilangan bulat yang menunjukkan batas kiri dan kanan dari rentang nilai sebuah partisi. Fungsi ini digunakan untuk membangun struktur awal dari struktur data *wavelet tree* itu sendiri yang algoritmanya telah dijelaskan pada Subbab 2.3. Pseudocode Fungsi Build ditunjukkan pada Gambar 3.3.

```
1  if partition.isEmpty()
2      return
3  mid = (left + right)/2
4  left_part = CreateLeftPartition(partition, mid)
5  right_part = CreateRightPartition(partition, mid)
6  map_left[node] = CreateMapping(partition, mid)
7  Build(left_part, node.left_child, left, mid)
8  Build(right_part, node.right_child, mid + 1, right)
```

Gambar 3.3: Pseudocode Fungsi Build I Love Kd-Trees

3.1.4 Desain Fungsi KthQuery

```
1  node = ROOT
2  left = 0
3  right = size − 1
4  while left < right
5      until_i = map_left[node][i]
6      mid = (left + right)/2
7      if k ≤ until_i
8          node = node.left_child
9          right = mid
10         i = until_i − 1
11     else
12         node = node.right_child
13         left = mid + 1
14         i = i − until_i
15         k = k − until_i
16 if l ≤ occurences[left].size()
17     return occurences[left][l − 1]
18 return −1
```

Gambar 3.4: Pseudocode Fungsi KthQuery

Fungsi ini akan menerima 3 parameter seperti yang telah dijelaskan pada Subbab 2.1.1 yaitu i, l, dan k dimana ketiganya merupakan bilangan bulat. Fungsi ini akan mengembalikan sebuah nilai yang merupakan dari jawaban dari pertanyaan pada subbab tersebut.

$node$ merupakan sebuah pointer yang menunjukkan *node* manaa yang sedang ditelusuri pada *wavelet tree* yang telah dibangun mula-mula nilainya berada pada *ROOT*. $left$ dan $right$ masing-masing merupakan sebuah bilangan bulat yang menunjukkan batas kiri dan

kanan dari rentang *node* sekarang, nilai awal dari kedua variabel ini adalah batas kiri dan kanan dari *root* yang kemudian akan berubah sesuai dengan algoritma penelurusan yang telah dibahas pada Subbab 2.3.4. $ocurrences$ adalah sebuah array bilangan bulat yang telah diinisialisasi pada Fungsi Main. Pseudocode dari Fungsi KthQuery ditunjukkan pada Gambar 3.1.4.

3.2 Permasalahan I Love Kd-Trees II

Pada permasalahan ini sistem akan dibangun untuk menyelesaikan 2 jenis operasi yaitu, Operasi Menghitung Jumlah Elemen yang Aktif dan Operasi Mengubah Status dari Sebuah Elemen.

3.2.1 Deskripsi Umum Sistem

Sampai dengan baris ke-10 *pseudocode* dari fungsi main untuk permasalahan ini memiliki penjelasan yang sama dengan subbab 3.1.1. Perbedaan hanya terletak pada array $status$ yang digunakan untuk menyimpan status terkini dari masing-masing elemen.

Selanjutnya sistem akan menerima masukan bilangan bulat $type$ untuk menentukkan operasi mana yang akan dikerjakan. Jika $type$ bernilai 0 maka Operasi Menghitung Jumlah Elemen yang Aktif akan menerima 3 masukan berupa bilangan bulat yaitu i, l, dan k kemudian sistem akan menampilkan luaran sebuah bulangan bulat yang menyatakan banyak elemen bernilai k yang aktif pada rentang $[i..l]$.

Operasi Mengubah Status dari Sebuah Elemen akan menerima sebuah bilangan bulat r yang kemudian sistem akan melakukan perubahan status elemen pada indeks r. Pseudocode Fungsi Main ditunjukkan pada Gambar 3.5.

```
1  let elements be a new dynamic array
2  let status[0..MAXN] be a new array
3  let map_left be a dynamic arrays of dynamic arrays
4  let symbols be a new dictionary
5  N = Input() // Number of elements
6  Q = Input() // Number of operations
7  for i = 0 to N − 1
8      elements[i] = Input()
9  Init(elements)
10 Build(elements, 1, 0, size)
11 for q = 0 to Q − 1
12     type = Input()
13     if type = 0
14         i = Input(), l = Input(), k = Input()
15         Print(ActiveElementRangeQuery(i, l, k))
16     else
17         r = Input()
18         ToggleElementStatus(r)
```

Gambar 3.5: Pseudocode Fungsi Main I Love Kd-Trees II

3.2.2 Desain Fungsi Init

Fungsi ini memiliki penjelasan yang sama dengan Subbab 3.1.2 dengan tambahan inisialisasi array $status$ menjadi aktif pada mula-mula. Pseudocode Fungsi Init ditunjukkan pada Gambar 3.6.

3.2.3 Desain Fungsi Build

Penjelasan fungsi ini mirip dengan penjelasan Subbab 3.1.3. Namun terdapat perbedaan saat pembentukan *node leaf* dimana pada fungsi ini dibentuk struktur Binary Indexed *Tree* (BIT) yang di-

```
1  status = ACTIVE
2  sorted = elements.copy()
3  sort(sorted)
4  for element ∈ elements
5      symbols.insert(element)
6  for i to N − 1
7      elements[i] = symbols[elements[i]]
8  map_left.assign(3 ∗ size)
```

Gambar 3.6: Pseudocode Fungsi Init I Love Kd-Trees II

gunakan untuk melakukan penghitungan jumlah elemen aktif pada rentang yang telah diberikan. Penjelasan lebih detil mengenai Fungsi BITUpdate terdapat pada Subbab 3.2.8. Pseudocode Fungsi Build ditunjukkan pada Gambar 3.7.

```
 1  if partition.isEmpty()
 2      sz = partition.size()
 3      map_left[node].assign(sz, 0)
 4      for i = 1 to partition.size()
 5          BITUpdate(map_left[node], sz, i, 1)
 6  mid = (left + right)/2
 7  left_part = CreateLeftPartition(partition, mid)
 8  right_part = CreateRightPartition(partition, mid)
 9  map_left[node] = CreateMapping(partition, mid)
10  Build(left_part, node.left_child, left, mid)
11  Build(right_part, node.right_child, mid + 1, right)
```

Gambar 3.7: Pseudocode Fungsi Build I Love Kd-Trees II

3.2.4 Desain Fungsi ActiveElementRangeQuery

Fungsi ini menerima 3 buah bilangan bulat sebagai parameter i, l, dan k, masing-masing telah dijelaskan pada Subbab 2.1.2. Fungsi ini akan mengeluarkan nilai balikan berupa hasil dari perhitungan dari Fungsi ActiveElementQuery dengan parameter l dan $i-1$. Penjelasan lebih lanjut pada Subbab 3.2.5. Pseudocode Fungsi ActiveElementRangeQuery ditunjukkan pada Gambar 3.8.

```
1  if symbols.find(k)
2      return ActiveElementQuery(l, symbols[k])−
                  ActiveElementQuery(i − 1, symbols[k])
3  else
4      return 0
```

Gambar 3.8: Pseudocode Fungsi ActiveElementRangeQuery

3.2.5 Desain Fungsi ActiveElementQuery

Fungsi ini menerima 2 buah bilangan bulat sebagai parameter i dan k. Fungsi ini nantinya akan mengembalikan sebuah nilai yang menyatakan banyak elemen k yang aktif pada rentang $[1 .. i]$. Algoritma untuk mendapatkan nilai ini dijelaskan pada Subbab 2.3.3.

$node$ merupakan sebuah pointer yang menunjukkan *node* manaa yang sedang ditelusuri pada *wavelet tree* yang telah dibangun mula-mula nilainya berada pada $ROOT$. $left$ dan $right$ masing-masing merupakan sebuah bilangan bulat yang menunjukkan batas kiri dan kanan dari rentang *node* sekarang, nilai awal dari kedua variabel ini adalah batas kiri dan kanan dari *root*.

Penjelasan detil pada algoritma pada Subbab **??**. Fungsi BITQuery akan dijelaskan pada Subbab 3.2.9. Pseudocode Fungsi ActiveEle-

mentQuery ditunjukkan pada Gambar 3.9.

```
1  node = ROOT
2  left = 0
3  right = size − 1
4  if x < 0
5      return 0
6  while left < right
7      mid = (left + right)/2
8      until_i = map_left[node][x]
9      if k ≤ mid
10         node = node.left_child
11         right = mid
12         x = until_i − 1
13     else
14         node = node.right_child
15         left = mid + 1
16         x = x − until_i
17     if x < 0
18         return 0
19     x = min(x + 1, map_left[node].size())
20     return BITQuery(map_left[node], x)
```

Gambar 3.9: Pseudocode Fungsi ActiveElementQuery

3.2.6 Desain Fungsi ToggleElementStatus

Fungsi ini menerima sebuah bilangan bulat r yang menyatakan indeks dari elemen yang akan diubah statusnya. Fungsi ini akan memanggil Fungsi ToggleActualStatus yang dijelaskan pada Subbab 3.2.7 untuk melakukan perubahan pada array $status$ serta mengubah struktur dari *wavelet tree* itu sendiri. Algoritma dari fungsi ini

dijelaskan pada Subbab 2.3.5.

$node$ merupakan sebuah pointer yang menunjukkan *node* manaa yang sedang ditelusuri pada *wavelet tree* yang telah dibangun mula-mula nilainya berada pada $ROOT$. $left$ dan $right$ masing-masing merupakan sebuah bilangan bulat yang menunjukkan batas kiri dan kanan dari rentang *node* sekarang, nilai awal dari kedua variabel ini adalah batas kiri dan kanan dari *root*. Pseudocode Fungsi ToggleElementStatus ditunjukkan pada Gambar 3.10.

```
 1  status = ToggleActualStatus(r)
 2  node = ROOT
 3  left = 0
 4  right = size − 1
 5  val = elements[r]
 6  while left < right
 7      mid = (left + right)/2
 8      until_i = map_left[node][r]
 9      if val ≤ mid
10          node = node.left_child
11          right = mid
12          r = until_i − 1
13      else
14          node = node.right_child
15          left = mid + 1
16          r = r − until_i
17  sz = map_left[node].size()
18  BITUpdate(map_left[node], sz, r + 1, status)
```

Gambar 3.10: Pseudocode Fungsi ToggleElementStatus

3.2.7 Desain Fungsi ToggleActualStatus

Fungsi ini menerima sebuah bilangan bulat r sebagai parameternya, kemudian melakukan perubahan status untuk elemen ke-r. Pseudocode Fungsi ToggleActualStatus ditunjukkan pada Gambar 3.11.

```
1  status[r] =!status[r]
2  return status[r]?1 : −1
```

Gambar 3.11: Pseudocode Fungsi ToggleActualStatus

3.2.8 Desain Fungsi BITUpdate

Fungsi ini menerima tiag buah parameter, sebuah array arr dan dua buah bilangan bulat id dan val. arr merupakan array dari struktur BIT yang ingin dilakukan perubahan nilai. id merupakan indeks dari elemen yang nilainya akan ditambahkan. Perlu diingat bahwa dalam struktur BIT ini hanya dapat dilakukan operasi penambahan. Pseudocode Fungsi BITUpdate ditunjukkan pada Gambar 3.12.

```
1  while id ≤ size(arr)
2        arr[id] = arr[id] + val
3        id = id + (id & (−id))
```

Gambar 3.12: Pseudocode Fungsi BITUpdate

3.2.9 Desain Fungsi BITQuery

Fungsi ini menerima duah parameter, sebuah array arr dan sebuah bilangan bulat id. arr merupakan array dari struktur BIT yang dii-

nginkan dan id merupakan batas kanan dari *query* yang diinginkan. Operasi *query* yang didukung oleh struktur data ini adalah penjumlahan dengan rentang $[1 .. id]$. Nilai ini sendiri merupakan nilai balikan dari fungsi ini. Pseudocode Fungsi BITQuery ditunjukkan pada Gambar 3.13.

```
1  ret = 0
2  while id > 0
3      ret = ret + arr[id]
4      id = id − (id & (−id))
5  return ret
```

Gambar 3.13: Pseudocode Fungsi BITQuery

3.3 Permasalahan I Love Kd-Trees III

Pada permasalahan ini sistem akan dibangun untuk menyelesaikan 2 jenis operasi yaitu, Operasi Mencari Bilangan Terkecil ke-K dan Operasi Menukar Elemen yang Bersebelahan.

3.3.1 Deskripsi Umum Sistem

Desain sistem untuk permasalahan ini merupakan pengembangan dari sistem untuk permasalahan I Love Kd-Trees yang telah dijelaskan pada Subbab 3.1.1. Operasi Menukar Elemen yang Bersebelahan akan ditambahkan pada permasalahan ini.

Array $bitvector$ menyimpan data boolean. Nilai pada indeks tertentu bernilai $true$ jika nilai pada elemen tersebut $\leq m_T(S)$ dan akan bernilai $false$ jika sebaliknya.

Array $rev\_occurence$ meyimpan indeks yang menyatakan posisi elemen ke-i pada array $occurences$.

Selanjutnya sistem akan menerima masukan bilangan bulat $type$ untuk menentukkan operasi mana yang akan dikerjakan. Operasi Mencari Bilangan Terkecil ke-K memiliki penjelasan yang sama pada subbab 3.1.1.

```
1  let elements be a new dynamic array
2  let map_left[ be a dynamic arrays of dynamic arrays
3  let bitvector be a dynamic arrays of dynamic arrays
4  let occurences[1..size] be an array of dynamic arrays
5  let rev_occurence[0..MAXN] be a new array
6  let symbols be a new dictionary
7  N = Input() // Number of elements
8  Q = Input() // Number of operations
9  for i = 0 to N - 1
10     elements[i] = Input()
11 Init(elements)
12 for i = 0 to N - 1
13     occurences[elements[i]].push(i)
14     rev_occurence[i] = occurences[elements[i]].size()
15 Build(elements, 1, 0, size)
16 for q = 0 to Q - 1
17     type = Input()
18     if type = 0
19         i = Input()l = Input()k = Input()
20         Print(KthQuery(i, l, k)
21     else
22         i = Input()
23         SwapElement(i)
24
```

Gambar 3.14: Pseudocode Fungsi Main I Love Kd-Trees III

Operasi Menukar Elemen yang Bersebelahan akan menerima sebuah bilangan bulat i sebagai masukan. Elemen pada indeks posisi ke i dan $i + 1$ akan ditukar posisinya. Pseudocode Fungsi Main ditunjukkan pada Gambar 3.14.

3.3.2 Desain Fungsi Init

Penjelasan fungsi ini sama dengan Fungsi Init untuk permasalahan I Love Kd-Trees yang telah dijelaskan pada Subbab 3.1.2. Perbedaan terdapat pada baris 8 khusus dibutuhkan untuk permasalahan ini. Pseudocode Fungsi Init ditunjukkan pada Gambar 3.15.

```
1  sorted = elements.copy()
2  sort(sorted)
3  for element ∈ elements
4      symbols.insert(element)
5  for i to N − 1
6      elements[i] = symbols[elements[i]]
7  map_left.assign(3 ∗ size)
8  bitvector.assign(3 ∗ size)
```

Gambar 3.15: Pseudocode Fungsi Init I Love Kd-Trees III

3.3.3 Desain Fungsi Build

Fungsi ini memiliki penjelasan yang sama dengan Subbab 3.1.3. Baris ke 7 sedikit berbeda dengan fungsi sebelumnya karena perlu dilakukan pembentukan dari struktur array $bitvector$. Pseudocode Fungsi Build ditunjukkan pada Gambar 3.16.

```
1  if partition.isEmpty()
2      return
3  mid = (left + right)/2
4  left_part = CreateLeftPartition(partition, mid)
5  right_part = CreateRightPartition(partition, mid)
6  map_left[node] = CreateMapping(partition, mid)
7  bitvector[node] = CreateBitvector(partition, mid)
8  Build(left_part, node.left_child, left, mid)
9  Build(right_part, node.right_child, mid + 1, right)
```

Gambar 3.16: Pseudocode Fungsi Build I Love Kd-Trees III

3.3.4 Desain Fungsi KthQuery

Fungsi ini sama persis dengan Fungsi KthQuery yang telah dijelaskan pada Subbab 3.1.4. Pseudocode Fungsi KthQuery ditunjukkan pada Gambar 3.4.

3.3.5 Desain Fungsi SwapElement

Fungsi ini menerima satu buah parameter bilangan bulat i yang menyatakan elemen pada posisi berapa yang akan ditukar dengan elemen di sebelah kanannya. Fungsi ini akan melakukan perubahan yang diperlukan untuk menjaga struktur dari *wavelet tree*.

Perubahan ini juga menyebabkan indeks kemunculan suatu angka juga berubah maka akan dilakukan perubahan pada array $occurences$ dan $rev\_occurence$ yang akan dijelaskan pada bagian Subbab 3.3.6.

$node$ merupakan sebuah pointer yang menunjukkan *node* manaa yang sedang ditelusuri pada *wavelet tree* yang telah dibangun mula-mula nilainya berada pada $ROOT$. $left$ dan $right$ masing-masing

merupakan sebuah bilangan bulat yang menunjukkan batas kiri dan kanan dari rentang *node* sekarang, nilai awal dari kedua variabel ini adalah batas kiri dan kanan dari *root* yang kemudian akan berubah sesuai dengan algoritma penelurusan yang telah dibahas pada Subbab 2.3.6. Selama penulurusan ini berlangsung juga akan dilakukan perubahan pada array map_left. Pseudocode Fungsi SwapElement ditunjukkan pada Gambar 3.17.

3.3.6 Desain Fungsi UpdateOccurence

Fungsi ini akan menerima sebuah parameter bilangan bulat i yang menyatakan pada posisi mana penukaran elemen akan dilakukan. Pada kondisi sebelumnya $occurence[x][y]$ menyimpan indeks kemunculan ke-y dari elemen x. Ketika penukaran dengan elemen pada indeks $i+1$ terjadi maka struktur tersebut hanya berubah ketika elemen pada indeks i dan $i+1$ merupakan elemen yang berbeda. Array $rev\_occurence$ merupakan struktur bantu yang menyimpan indeks dari elemen ke-i pada array $occurence[x]$. Pseudocode Fungsi UpdateOccurence ditunjukkan pada Gambar 3.18.

```
1  if elements[i] ≠ elements[i + 1]
2      j = i + 1
3      occurences[elements[i]][rev_occurence[i]] = j
4      occurences[elements[j]][rev_occurence[j]] = i
5      swap(elements[i], elements[j])
6      swap(rev_occurence[i], rev_occurence[j])
```

Gambar 3.18: Pseudocode Fungsi UpdateOccurence

```
1   UpdateOccurence(i)
2   node = ROOT
3   left = 0
4   right = size − 1
5   while left < right
6       if bitvector[node][i] ≠ bitvector[node][i + 1]
7           if !bitvector[node][i]
8               map_left[node][i]++
9           else
10              map_left[node][i]--
11          swap(bitvector[node][i],
                    bitvector[node][i + 1])
12          return
13      else
14          mid = (left + right)/2
15          until_i = map_left[node][i]
16          if bitvector[node][i])
17              node = node.left_child
18              i = until_i − 1
19              right = mid
20          else
21              node = node.right_child
22              i = i − until_i
23              left = mid + 1
```

Gambar 3.17: Pseudocode Fungsi SwapElement

Halaman ini sengaja dikosongkan

BAB 4
IMPLEMENTASI

Pada bab ini akan dijelaskan implementasi dari algoritma dan struktur data berdasarkan desain yang telah dilakukan.

4.1 Lingkungan Implementasi

Lingkungan implementasi dan pengembangan yang dilakukan adalah sebagai berikut:

1. Perangkat Keras
    - Processor Intel Core i3-3217U CPU @ 1.80GHz x 4.
    - *Memory* 9.8 GB.
2. Perangkat Lunak
    - Sistem operasi Linux Mint 17.1 Rebecca KDE 64 bit.
    - Sublime Text 3.

4.2 Permasalahan I Love Kd-Trees

Subbab ini akan menjelaskan implementasi dari algoritma dan struktur data untuk menyelesaikan permasalahan I Love Kd-Trees.

4.2.1 Implementasi Fungsi Main

Fungsi Main diimplementasikan sesuai dengan pseudocode 3.1.1. Mula-mula fungsi ini membaca masukan dengan menggunakan fungsi scanf, membangun struktur data *wavelet tree* kemudian menjawab *query* berdasarkan parameter yang diberikan. Luaran yang dihasilkan adalah hasil dari operasi $quantile$. Implementasi dari Fungsi Main dapat dilihat pada Kode Sumber 4.1.

```
1  int main() {
2    int n, q;
3    scanf("%d%d", &n, &q);
4    for (int i = 0; i < n; i++) {
5      scanf("%d", &elements[i]);
6    }
7    init();
8    build(elements, 1, 0, symbols.size() - 1);
9    for (int i = 0; i < n; i++) {
10     occurences[elements[i]].emplace_back(i);
11   }
12   while (q--) {
13     int k, i, l;
14     scanf("%d%d%d", &k, &i, &l);
15     printf("%d\n", kthQuery(i, l, k));
16   }
17   return 0;
18 }
```

Kode Sumber 4.1: Implementasi Fungsi Main

4.2.2 Implementasi Variabel Global

Variabel digunakan untuk mempermudah masing-masing fungsi untuk mengakses data yang sama. Penggunaan variabel global pada implementasi ini diterapkan pada $elements$ yang menyimpan data masukan, map_left yang menyimpan struktur dari *wavelet tree* itu sendiri, variabel $symbols$ yang menyimpan simbol-simbol yang muncul pada data, serta $occurences$ yang menyimpan index kemunculan untuk masing-masing symbols. Implementasi dari Variabel Global dapat dilihat pada Kode Sumber 4.2.

```
1  vector<int> elements;
2  vector<vector<int> > map_left;
3  vector<int> occurences[100002];
4  unordered_map<int, int> symbols;
```

Kode Sumber 4.2: Implementasi Variabel Global

4.2.3 Implementasi Fungsi Init

Pada awalnya fungsi ini akan membangun sebuah array baru yang disimpan pada variabel sorted. Tujuannya adalah untuk membentuk simbol-simbol yang ada pada data secara terurut tanpa mengubah data awal. Dilanjutkan dengan menginisialisasi variabel-variabel global yang dibutuhkan. Implementasi dari Fungsi Init dapat dilihat pada Kode Sumber 4.3.

```
1  void init() {
2    vector<int> sorted(elements);
3    sort(sorted.begin(), sorted.end());
4    for (int i = 0; i < elements.size(); i++) {
5      symbols.insert({sorted[i], symbols.size()});
6    }
7    for (int i = 0; i < elements.size(); i++) {
8      elements[i] = symbols[elements[i]];
9    }
10   for (int i = 0; i < 3 * symbols.size(); i++) {
11     map_left.emplace_back(vector<int>());
12   }
13 }
```

Kode Sumber 4.3: Implementasi Fungsi Init

4.2.4 Implementasi Fungsi Build

Fungsi ini akan membentuk struktur awal dari struktur data *wavelet tree*. Algoritma pembentukan secara detil telah dijelaskan pada Subbab 2.3 dengan desain yang telah dibuat pada Subbab 3.1.3. Fungsi CreateRightPartition(), CreateLeftPartition(), dan CreateMapping() dijabarkan pada baris 6 sampai 15.

Array dinamis arr diimplementasikan menggunakan *C++ STL(Standard* Template *Library)* yaitu vector. Sedangkan untuk menyimpan *node* pada *wavelet tree* diimplementasikan menggunakan sebuah array dimana *node root* terletak pada posisi 1,

kemudian anak kiri terletak pada indeks $node * 2$, dan anak kanan terletak pada indeks $node * 2 + 1$. Hal ini dimungkinkan karena struktur dari *wavelet tree* menyerupai complete binary *tree*.

Strategi pengindeksan ini akan digunakan juga untuk permasalahan I Love Kd-Trees II dan III. Implementasi dari Fungsi Build dapat dilihat pada Kode Sumber 4.4.

```
1  void build(vector<int> &arr, int id,\
2             int left, int right) {
3    if (left == right) return;
4
5    int mid = (left + right) >> 1;
6    vector<int> arr_left, arr_right;
7
8    for (int i = 0; i < arr.size(); i++) {
9      int temp = i ? map_left[id].back() : 0;
10     if (arr[i] <= mid) {
11       temp++;
12       arr_left.emplace_back(arr[i]);
13     } else {
14       arr_right.emplace_back(arr[i]);
15     }
16     map_left[id].emplace_back(temp);
17   }
18
19   build(arr_left, id << 1, left, mid);
20   build(arr_right, (id << 1) | 1, mid + 1, right);
21 }
```

Kode Sumber 4.4: Implementasi Fungsi Build

4.2.5 Implementasi Fungsi KthQuery

Fungsi ini merupakan implementasi dari desain yang telah dijelaskan pada Subbab 3.1.4. Fungsi ini akan melakukan penulusuran pada *wavelet tree* yang telah dibangun untuk menemukan elemen terkecil ke-k dan memberikan nilai kembalian berupa indeks kemuncul-

an ke-l dari elemen tersebut pada array awal. Implementasi dari Fungsi KthQuery dapat dilihat pada Kode Sumber 4.5.

```
1  int kthQuery(int i, int l, int k) {
2    int id = 1;
3    int left = 0;
4    int right = symbols.size() - 1;
5
6    while (left < right) {
7      int until_i = map_left[id][i];
8      int mid = (left + right) >> 1;
9      id <<= 1;
10     if (k <= until_i) {
11       right = mid;
12       i = until_i - 1;
13     } else {
14       id++;
15       left = mid + 1;
16       i -= until_i;
17       k -= until_i;
18     }
19   }
20   if (l <= occurences[left].size())
21     return occurences[left][l - 1];
22   return -1;
23 }
```

Kode Sumber 4.5: Implementasi Fungsi KthQuery

4.3 Permasalahan I Love Kd-Trees II

Subbab ini akan menjelaskan implementasi dari algoritma dan struktur data untuk menyelesaikan permasalahan I Love Kd-Trees II.

4.3.1 Implementasi Fungsi Main

Fungsi Main pada permasalahan ini memiliki implementasi yang mirip dengan Fungsi Main pada Subbab 4.2.1. Namun berbeda

saat mengatasi operasi-operasi yang ada untuk permasalahan ini. Implementasi dari Fungsi Main dapat dilihat pada Kode Sumber 4.6.

```
1  int main() {
2    int n, q;
3    scanf("%d%d", &n, &q);
4    for (int i = 0; i < n; i++) {
5      scanf("%d", &elements[i]);
6    }
7    init();
8    build(elements, 1, 0, symbols.size() - 1);
9    while (q--) {
10     int type;
11     getnum<int>(type);
12     if (!type) {
13       int i, l, k;
14       scanf("%d%d%d", &i, &l, &k);
15       printf("%d\n",\
16         activeElementRangeQuery(i, l, k));
17     } else {
18       int r;
19       scanf("%d", &r);
20       toggleElementStatus(r);
21     }
22   }
23   return 0;
24 }
```

Kode Sumber 4.6: Implementasi Fungsi Main

4.3.2 Implementasi Variabel Global

Variabel global yang diimplementasikan pada permasalahan ini adalah array dinamis $elements$, $status$, map_left, dan $symbols$. Penggunaan array $elements$, map_left, dan $symbols$ memiliki penjelasan yang sama dengan Subbab 4.2.2. Array $status$ digunakan untuk menyimpan status terkini dari data pada masing-masing

indeks. Implementasi dari variabel global dapat dilihat pada Kode Sumber 4.7.

```
1  vector<int> elements;
2  vector<bool> status;
3  vector<vector<int> > map_left;
4  unordered_map<int, int> symbols;
```

Kode Sumber 4.7: Implementasi Variabel Global

4.3.3 Implementasi Fungsi Init

Fungsi ini memiliki implementasi yang hampir sama dengan Subbab 4.2.3. Namun pada permasalahan ini membutuhkan sebuah array tambahan, $status$, yang menyimpan status terkini dari masing-masing elemen. Mula-mula diinisialisasi dengan nilai $true$ atau 1. Implementasi dari Fungsi Init dapat dilihat pada Kode Sumber 4.8.

```
1  void init() {
2    status.assign(n, 1);
3
4    vector<int> sorted(elements);
5    sort(sorted.begin(), sorted.end());
6
7    for (int i = 0; i < elements.size(); i++) {
8      symbols.insert({sorted[i], symbols.size()});
9    }
10
11   for (int i = 0; i < elements.size(); i++) {
12     elements[i] = symbols[elements[i]];
13   }
14
15   for (int i = 0; i < 3 * symbols.size(); i++) {
16     map_left.emplace_back(vector<int>());
17   }
18 }
```

Kode Sumber 4.8: Implementasi Fungsi Init

4.3.4 Implementasi Fungsi Build

Implementasi dari Fungsi Build ini memiliki kemiripan dengan implementasi Fungsi Build pada Subbab 4.2.4. Perbedaannya terletak pada saat mencapai *node leaf*. Pada permasalahan ini dibutuhkan cara untuk menghitung elemen yang aktif pada suatu rentang, oleh karena itu diimplementasikan struktur data BIT untuk membantu menyelesaikan permasalahan ini. Implementasi dari Fungsi Build dapat dilihat pada Kode Sumber 4.9.

```
1  void build(vector<int> &arr, int id,\
2             int left, int right) {
3    if (left == right) {
4      map_left[id].assign(arr.size() + 1, 0);
5      for (int i = 0; i < arr.size(); i++) {
6        BITUpdate(map_left[id],\
7          map_left[id].size() - 1, i + 1, 1);
8      }
9      return;
10   }
11   int mid = (left + right) >> 1;
12   vector<int> arr_left, arr_right;
13   for (int i = 0; i < arr.size(); i++) {
14     int temp = i ? map_left[id].back() : 0;
15     if (arr[i] <= mid) {
16       temp++;
17       arr_left.emplace_back(arr[i]);
18     } else {
19       arr_right.emplace_back(arr[i]);
20     }
21     map_left[id].emplace_back(temp);
22   }
23   build(arr_left, id << 1, left, mid);
24   build(arr_right, (id << 1) | 1, mid + 1, right);
25 }
```

Kode Sumber 4.9: Implementasi Fungsi Build

4.3.5 Implementasi Fungsi ActiveElementRangeQuery

Fungsi ActiveElementRangeQuery merupakan fungsi sederhana yang membantu untuk menyelesaikan permasalahan Operasi Menghitung Jumlah Elemen yang Aktif.

Operasi tersebut memiliki rentang pencarian dari $[l \,..\, r]$ yang diselesaikan dengan melakukan pencarian dengan rentang $[1 \,..\, r]$ dikurangi dengan rentang $[1 \,..\, l-1]$. Dimana pencarian tersebut diimplementasikan pada Subbab 4.3.6. Implementasi dari Fungsi ActiveElementRangeQuery dapat dilihat pada Kode Sumber 4.10.

```
1  int activeElementRangeQuery(int i, int l, int k) {
2    unordered_map<int, int>::iterator\
3     it = symbols.find(k);
4    if (it == symbols.end())
5      return 0;
6    return activeElementQuery(l, it->second) -\
7      activeElementQuery(i - 1, it->second);
8  }
```

Kode Sumber 4.10: Implementasi Fungsi ActiveElementRangeQuery

4.3.6 Implementasi Fungsi ActiveElementQuery

Implementasi dari Fungsi ActiveElementQuery didasarkan pada pseudocode 3.9. Fungsi ini akan melakukan proses penulusuran pada struktur data *wavelet tree* yang telah dibangun untuk menemukan *node* yang mewakili symbol k pada rentang $[1 \,..\, x]$. Pada *node leaf* yang ditemukan dilakukan proses penghitungan elemen yang aktif pada struktur data BIT yang telah dibangun sebelumnya. Algoritma detil untuk menyelesaikan permasalahan ini telah dijelaskan pada Subbab 2.3.3.

Implementasi dari Fungsi ActiveElementRangeQuery dapat dilihat pada Kode Sumber 4.11.

```
1  int activeElementQuery(int x, int k) {
2    int id = 1;
3    int left = 0;
4    int right = symbols.size() - 1;
5    if (x < 0)
6      return 0;
7    while (left < right) {
8      int mid = (left + right) >> 1;
9      int until_i = map_left[id][x];
10     id <<= 1;
11     if (k <= mid) {
12       right = mid;
13       x = until_i - 1;
14     } else {
15       id++;
16       left = mid + 1;
17       x -= until_i;
18     }
19     if (x < 0) return 0;
20   }
21   x = min(x + 1, (int)map_left[id].size());
22   return BITQuery(map_left[id], x);
23 }
```

Kode Sumber 4.11: Implementasi Fungsi ActiveElementQuery

4.3.7 Implementasi Fungsi ToggleElementStatus

Fungsi ini akan memanggil Fungsi ToggleActualStatus yang akan mengubah status elemen ke-r nilai kembalian dari fungsi tersebut akan digunakan untuk mengubah data pada BIT sesuai dengan status sekarang. Fungsi ini akan dijelaskan pada Subbab 4.3.8 Selanjutnya fungsi akan melakukan penulusuran pada *wavelet tree* untuk mencari *node* yang menyimpan data elemen pada indeks ke-r. Setelah *node leaf* ditemukan maka akan dilakukan *update* pada struktur data BIT. Implementasi dari Fungsi ToggleElementStatus dapat dilihat pada Kode Sumber 4.12.

```
1  void toggleElementStatus(int r) {
2    int id = 1;
3    int left = 0;
4    int right = symbols.size() - 1;
5    int val = elements[r];
6    int status = toggleActualStatus(r);
7    while (left < right) {
8      int mid = (left + right) >> 1;
9      int until_i = map_left[id][r];
10     id <<= 1;
11     if (val <= mid) {
12       right = mid;
13       r = until_i - 1;
14     } else {
15       id++;
16       left = mid + 1;
17       r -= until_i;
18     }
19   }
20   BITUpdate(map_left[id], map_left[id].size(),\
21     r + 1, status);
22 }
```

Kode Sumber 4.12: Implementasi Fungsi ToggleElementStatus

4.3.8 Implementasi Fungsi ToggleActualStatus

Fungsi ini akan memberikan nilai kembalian bernilai -1 jika status elemen pada indeks ke-r berubah menjadi non-aktif atau 1 jika status elemen pada indeks ke-r menjadi aktif. Implementasi dari Fungsi ToggleActualStatus dapat dilihat pada Kode Sumber 4.13.

```
1  int toggleActualStatus(int r) {
2    status[r] = !status[r];
3    return status[r] ? 1 : -1;
4  }
```

Kode Sumber 4.13: Implementasi Fungsi Toggle Actual Status

4.3.9 Implementasi Fungsi BITUpdate

Pembagian tanggung jawab pada struktur data BIT ini dapat dicapai dengan menambahkan nilai LSB(Least Significant Bit) pada representasi binary dari indeks sekarang. Indeks pada kondisi tertentu disimpan pada variabel id. Implementasi dari Fungsi BITUpdate dapat dilihat pada Kode Sumber 4.14.

```
1  void BITUpdate(vector<int> &arr, int MAXN,\
2    int id, int val) {
3    for (; id <= MAXN; id += id & (-id)) {
4      arr[id] += val;
5    }
6  }
```

Kode Sumber 4.14: Implementasi Fungsi BIT *Update*

4.3.10 Implementasi Fungsi BITQuery

Fungsi ini akan mengembalikan nilai kumulatif dari indeks $[1 .. id]$ untuk struktur array BIT yang diberikan. Implementasi dari Fungsi BITQuery dapat dilihat pada Kode Sumber 4.15.

```
1  int BITQuery(vector<int> &arr, int id) {
2    int ret = 0;
3    for (; id > 0; id -= id & (-id)) {
4      ret += arr[id];
5    }
6    return ret;
7  }
```

Kode Sumber 4.15: Implementasi Fungsi BIT *Query*

4.4 Permasalahan I Love Kd-Trees III

Subbab ini akan menjelaskan implementasi dari algoritma dan struktur data penyelesaian permasalahan I Love Kd-Trees III.

4.4.1 Implementasi Fungsi Main

Fungsi ini mula-mula akan menerima masukan yang akan menjadi awal, membangun struktur data *wavelet tree* dan kemudian melakukan operasi yang telah didefinisikan sebelumnya. Terdapat dua operasi yaitu, Operasi Mencari Bilangan Terkecil ke-K dan Operasi Menukar Elemen yang Bersebelahan. Implementasi dari Fungsi Main dapat dilihat pada Kode Sumber 4.16.

```
1  int main() {
2    int n, q;
3    scanf("%d%d", &n, &q);
4    for (int i = 0; i < n; i++) {
5      scanf("%d", &elements[i]);
6    }
7    init();
8    build(elements, 1, 0, symbols.size() - 1);
9    for (int i = 0; i < n; i++) {
10     occurences[elements[i]].emplace_back(i);
11     rev_occurence.push_back(
12       occurences[elements[i]].size() - 1);
13   }
14   while (q--) {
15     int type;
16     getnum<int>(type);
17     if (!type) {
18       int i, l, k;
19       scanf("%d%d%d", &i, &l, &k);
20       printf("%d\n", kthQuery(i, l, k));
21     } else {
22       int i;
23       scanf("%d", &i);
24       swapElement(i);
25     }
26   }
27   return 0;
28 }
```

Kode Sumber 4.16: Implementasi Fungsi Main

4.4.2 Implementasi Variabel Global

Variabel global yang ada pada implementasi ini secara garis besar mirip dengan Subbab 4.2.2. Pada implementasi ini ditambahkan array *bitvector* dan *rev_occurence* yang telah dijelaskan kegunaannya pada Subbab 3.3.1. Implementasi dari variable global dapat dilihat pada Kode Sumber 4.17.

```
1  vector<int> elements, rev_occurences
2  vector<vector<int> > map_left;
3  vector<vector<bool> > bitvector;
4  vector<int> occurences[1000002];
5  unordered_map<int, int> symbols;
```

Kode Sumber 4.17: Implementasi Variabel Global

4.4.3 Implementasi Fungsi Init

Fungsi ini memiliki implementasi yang hampir sama dengan Subbab 4.2.3. Baris ke-12 ditambahkan untuk menginisialisasi array *bitvector*. Implementasi dari Fungsi Init dapat dilihat pada Kode Sumber 4.18.

```
1  void init(s) {
2    vector<int> sorted(elements);
3    sort(sorted.begin(), sorted.end());
4    for (int i = 0; i < elements.size(); i++) {
5      symbols.insert({sorted[i], symbols.size()});
6    }
7    for (int i = 0; i < elements.size(); i++) {
8      elements[i] = symbols[elements[i]];
9    }
10   for (int i = 0; i < 3 * symbols.size(); i++) {
11     map_left.emplace_back(vector<int>());
12     bitvector.emplace_back(vector<bool>());
13   }
14 }
```

Kode Sumber 4.18: Implementasi Fungsi Init

4.4.4 Implementasi Fungsi Build

Fungsi ini diimplementasikan berdasarkan desain 3.3.3. Pada baris 6 hingga 17 merupakan penjabaran dari fungsi CreateRightPartition(), CreateLeftPartition(), CreateMapping(), dan CreateBitVector(). Kemudian fungsi akan dilanjutkan secara rekursif untuk membentuk struktur anak kiri dan anak kanan dari *node* tersebut. Implementasi dari Fungsi Build dapat dilihat pada Kode Sumber 4.19.

```
1  void build(vector<int> &arr, int id,\
2             int left, int right) {
3    if (left == right) return;
4    int mid = (left + right) >> 1;
5    vector<int> arr_left, arr_right;
6    for (int i = 0; i < arr.size(); i++) {
7      int temp = i ? map_left[id].back() : 0;
8      if (arr[i] <= mid) {
9        temp++;
10       arr_left.emplace_back(arr[i]);
11       bitvector[id].emplace_back(1);
12     } else {
13       arr_right.emplace_back(arr[i]);
14       bitvector[id].emplace_back(0);
15     }
16     map_left[id].emplace_back(temp);
17   }
18   build(arr_left, id << 1, left, mid);
19   build(arr_right, (id << 1) | 1, mid + 1, right);
20 }
```

Kode Sumber 4.19: Implementasi Fungsi Build

4.4.5 Implementasi Fungsi KthQuery

Fungsi ini memiliki implementasi yang sama dengan Subbab 4.2.5. Implementasi dari Fungsi KthQuery dapat dilihat pada Kode Sumber 4.5.

4.4.6 Implementasi Fungsi SwapElement

Fungsi ini akan memanggil Fungsi UpdateOccurence terlebih dahulu untuk mengubah indeks kemunculan dari elemen yang akan ditukar. Kemudian fungsi ini akan melakukan perubahan pada array map_left dan array $bitvector$ sesuai dengan algoritma yang telah dijelaskan pada Subbab 2.3.6. Implementasi dari Fungsi SwapElement dapat dilihat pada Kode Sumber 4.20.

```
1  void swapElement(int i) {
2    updateOccurence(i);
3    int left = 0, right = symbols.size() - 1;
4    int id = 1;
5    while (left < right) {
6      if (bitvector[id][i] != bitvector[id][i + 1]) {
7        if (!bitvector[id][i]) {
8          map_left[id][i]++;
9        } else {
10         map_left[id][i]--;
11       }
12       swap(bitvector[id][i], bitvector[id][i + 1]);
13       break;
14     } else {
15       int mid = (left + right) >> 1;
16       int until_i = map_left[id][i];
17       if (bitvector[id][i]) {
18         i = until_i - 1;
19         id <<= 1;
20         right = mid;
21       } else {
22         i -= until_i;
23         id <<= 1;
24         id |= 1;
25         left = mid + 1;
26       }
27     }
28   }
29 }
```

Kode Sumber 4.20: Implementasi Fungsi SwapElement

4.4.7 Implementasi Fungsi UpdateOccurence

Implementasi fungsi ini mengikuti desain pada Subbab 3.3.6. Implementasi dari Fungsi UpdateOccurence dapat dilihat pada Kode Sumber 4.21.

```
1  void updateOccurence(int i) {
2    if (elements[i] != elements[i + 1]) {
3      occurences[elements[i]]
4        [rev_occurence[i]] = i + 1;
5      occurences[elements[i + 1]]
6        [rev_occurence[i + 1]] = i;
7      swap(rev_occurence[i], rev_occurence[i + 1]);
8      swap(elements[i], elements[i + 1]);
9    }
10 }
```

Kode Sumber 4.21: Implementasi Fungsi UpdateOccurence

Halaman ini sengaja dikosongkan

BAB 5

UJI COBA DAN EVALUASI

Pada bab ini akan dijelaskan tentang uji coba dan evaluasi dari implementasi sistem yang telah dilakukan pada bab 4.

5.1 Lingkungan Uji Coba

Lingkungan uji coba menggunakan sebuah komputer dengan spesifikasi perangkat lunak dan perangkat keras sebagai berikut:

1. Perangkat Keras
    - Processor Intel Core i3-3217U CPU @ 1.80GHz x 4.
    - *Memory* 9.8 GB.
2. Perangkat Lunak
    - Sistem operasi Linux Mint 17.1 Rebecca KDE 64 bit.
    - Sublime Text 3.

5.2 Uji Coba

5.2.1 Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan analisis penyelesaian sebuah contoh kasus menggunakan pendekatan penyelesaian yang telah dijelaskan pada subbab 2.3 serta pengumpulan berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ. Permasalahan yang diselesaikan adalah permasalahan I Love Kd-Trees dengan kode ILKQUERY, permasalahan I Love Kd-Trees II dengan kode ILKQUERY2, dan permasalahan I Love Kd-Trees III dengan kode ILKQUERYIII.

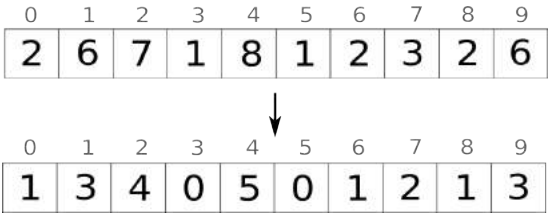
5.2.1.1 Permasalahan I Love Kd-Trees

Kasus yang akan digunakan sebagai bahan uji kebenaran dalam analisis penyelesaian permasalahan I Love Kd-Trees menggunakan contoh kasus sebagai berikut:

```
10 3
2 6 7 1 8 1 2 3 2 6
2 4 2
2 6 3
3 3 2
```

Gambar 5.1: Contoh kasus uji permasalahan I Love Kd-Trees.

Sebelum masuk ke dalam pembentukan *wavelet tree* yang sesuai untuk setiap permasalahan terlebih dahulu dilakukan pembuatan simbol yang muncul pada data awal. Dengan alasan simplisitas simbol yang dibuat menggunakan representasi bilangan bulat. Pembentukan simbol dari data awal dapat dilihat pada Gambar 5.2.

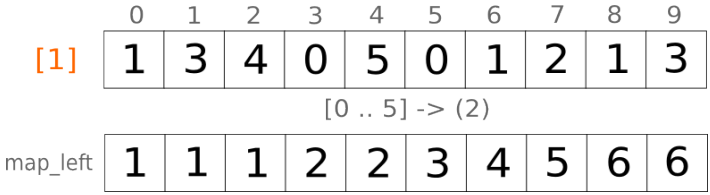


Gambar 5.2: Pembentukan *root wavelet tree*.

Mula-mula dalam setiap permasalahan adalah membangun struktur *wavelet tree* yang sesuai dengan masing-masing permasalahan. Telah dijelaskan pada Subbab 2.3 bahwa operasi dasar pada sebuah *wavelet tree* adalah map_left, operasi ini juga yang memberikan struktur pada *wavelet tree*. Struktur dari diimplementasikan dalam sebuah array dua dimensi yang masing-masing indeks menunjukkan

posisi *node* dan nilai dari $map\_left_T(i)$. Dimana T adalah sembarang *node* pada *wavelet tree* dan i adalah sebuah indeks dalam *node* tersebut.

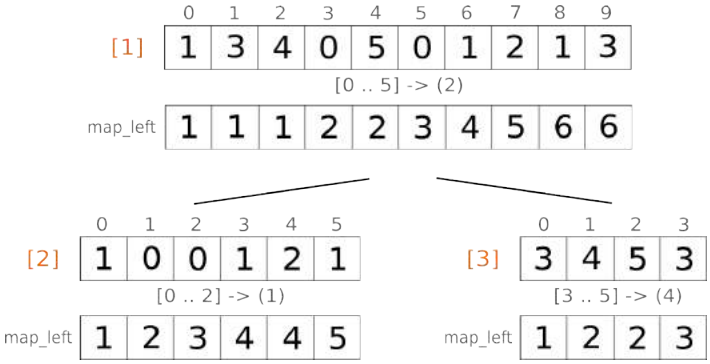
Mula-mula struktur *wavelet tree* akan dibangun pada *node root* dengan indeks 1. Pembentukan ini dapat dilihat pada Gambar 5.3. Nilai pada tanda kurung siku berwarna oranye menyatakan indeks dari *node* tersebut. Nilai pada tanda kurung siku berwarna abu-abu menyatakan rentang nilai yang menjadi tanggung jawab *node* tersebut dan nilai pada tanda kurung menyatakan $m_T(S)$ atau nilai tengah dari rentang nilai *node* tersebut.



Gambar 5.3: Pembentukan *root wavelet tree*.

Dikarenakan nilai rentang belum mencakup hanya satu nilai maka proses pembentukan dilanjutkan untuk membentuk anak kanan dan kiri dari *root*. Masing-masing akan diberi indeks 2 dan 3. Pembentukan anak dari *root* ditunjukkan pada Gambar 5.4. Rentang dari anak kiri *root* diperoleh dari rentang kiri hingga nilai tengah dari *root* sedangkan rentang dari anak kanan diperoleh dari nilai tengah + 1 hingga rentang kanan dari *root*. Partisi kiri akan bertanggung jawab terhadap semua nilai yang lebih kecil atau sama dengan nilai tengah dari *root*. Sebaliknya partisi kanan akan bertanggung jawab terhadapa semua nilai yang lebih besar dari nilai tengah *root*.

Pembentukan dilanjutkan untuk tingkat ke-3 dari *wavelet tree* ini. Gambar 5.5 merupakan pembentukan untuk anak kiri dari *root*. Struktur akhir dari *wavelet tree* ditunjukkan pada Gambar 5.6.



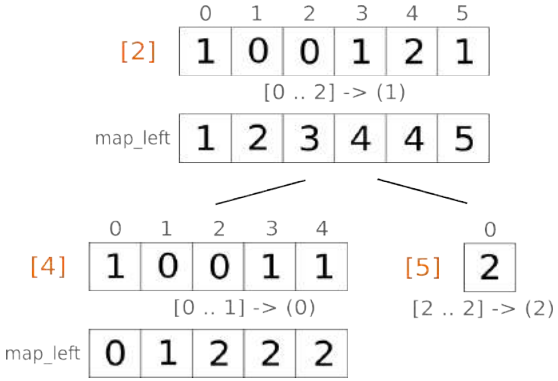
Gambar 5.4: Pembentukan *wavelet tree* pada tingkat 2.

val \ l	1	2	3
0	3	5	–
1	0	6	8
2	7	–	–
3	1	9	–
4	2	–	–
5	4	–	–

Tabel 5.1: Indeks kemunculan suatu simbol.

Dikarenakan pada permasalahan ini dibutuhkan tabel indeks kemunculan maka akan dibangun tabel indeks kemunculan untuk masing-masing simbol. Tabel tersebut dibangun dengan menyimpan masing-masing indeks kemunculan simbol pada array 2 dimensi. Dimensi pertama menyatakan simbol itu sendiri sedangkan dimensi kedua menyatakan kemunculan ke-x dari simbol tersebut. Sehingga array $occurences[i][j]$ menyimpan indeks kemunculan ke-j dari simbol i. Tabel 5.1 menunjukkan struktur dari tabel tersebut.

Query pertama memiliki parameter $k = 2, i = 4, l = 2$. Tahapan

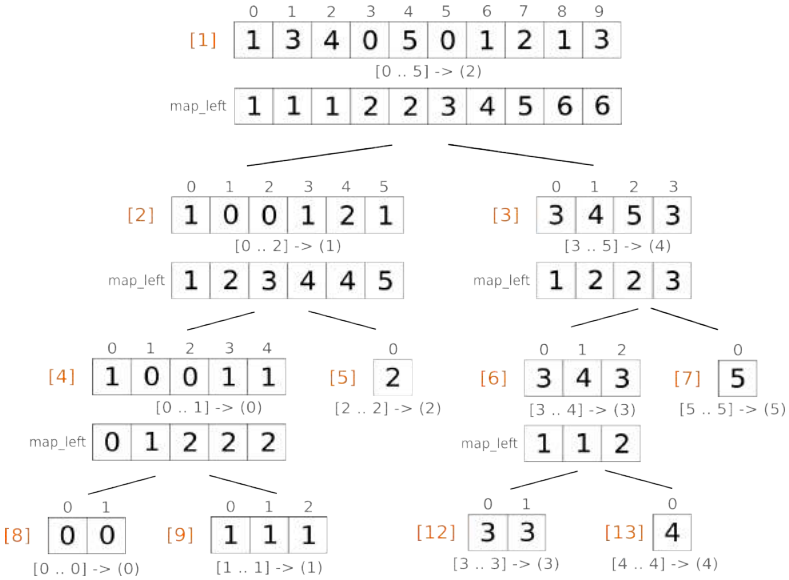


Gambar 5.5: Pembentukan *wavelet tree* pada tingkat 3.

pertama yaitu dengan mencari elemen terkecil ke-k dengan rentang pencarian $[1..i]$. Untuk memperoleh ini dilakukan penulusuran sesuai dengan algoritma penulusuran yang telah dijelaskan pada Subbab 2.3.4.

Dimulai dari *node root* dengan $id = 1$, $i = 4$, $left = 0$, dan $right = 5$, kegunaan masing-masing variabel telah dibahas pada Subbab 3.1.4. Kemudian dilanjutkan penulusuran ke anak kiri karena $k \leq map\_left[id][i]$, parameter-parameter tersebut menjadi $id = 2$, $i = map\_left[id][i] - 1 = 1$, $left = 0$, dan $right = 2$. Selanjutnya penulusuran masih dilanjutkan ke anak kiri karena $k \leq map\_left[id][i]$ dan parameter menjadi $id = 4$, $i = 1$, $left = 0, right = 1$. Pada penulusuran ke-3 akan bergerak ke anak kanan karena $k > map\_left[id][i]$ sehingga parameter menjadi $id = 9, i = i - map\_left[id][i] = 1, k = k - map\_left[id][i] = 1$, $left = 1, right = 1$. Pencarian telah mencapai *node leaf* dimana $left == right$ maka simbol terkecil ke-2 dari rentang $[1..4]$ adalah simbol 1.

Untuk menjawab pertanyaan dari operasi ini dapat dilihat pada tabel $occurences[1][2]$ dimana jawabannya adalah 6.



Gambar 5.6: Struktur *wavelet tree* yang terbentuk.

Query kedua memiliki parameter $k = 2$, $i = 6$, $l = 3$. Proses pencarian dan perubahan parameter adalah sebagai berikut,

$id = 1, i = 6, k = 2, left = 0, right = 5, map\_left[id][i] = 4;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri;

$id = 2, i = 3, k = 2, left = 0, right = 2, map\_left[id][i] = 4;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri;

$id = 4, i = 3, k = 2, left = 0, right = 1, map\_left[id][i] = 2;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri yang merupakan *node leaf*;

$d = 0$

Dikarenakan simbol 0 hanya muncul sebanyak 2 kali maka jawaban untuk *query* ini adalah $ans = -1$.

Query ketiga memiliki parameter $k = 3$, $i = 3$, $l = 2$. Proses pencarian dan perubahan parameter adalah sebagai berikut,

$id = 1, i = 3, k = 3, left = 0, right = 5, map\_left[id][i] = 2;$
$k > map\_left[id][i];$
dilanjutkan ke anak kanan;

$id = 3, i = 1, k = 1, left = 3, right = 5, map\_left[id][i] = 2;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri;

$id = 6, i = 1, k = 1, left = 3, right = 4, map\_left[id][i] = 1;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri yang merupakan *node leaf*;

$d = 3$

Maka jawaban untuk *query* keenam ini adalah $ans = 9$.

Secara berturut-turut jawaban dari ketiga *query* tersebut adalah 6, -1, dan 9. Kemudian sistem penyelesaian dijalankan dan diberi masukan sesuai kasus uji dari analisis sebelumnya dan hasil luaran sistem adalah 6, -1, dan 9 seperti yang terlihat pada Gambar 5.7.

```
10 3
2 6 7 1 8 1 2 3 2 6
2 4 2
6
2 6 3
-1
3 3 2
9
```

Gambar 5.7: Hasil luaran program pada contoh kasus uji ILKQUERY.

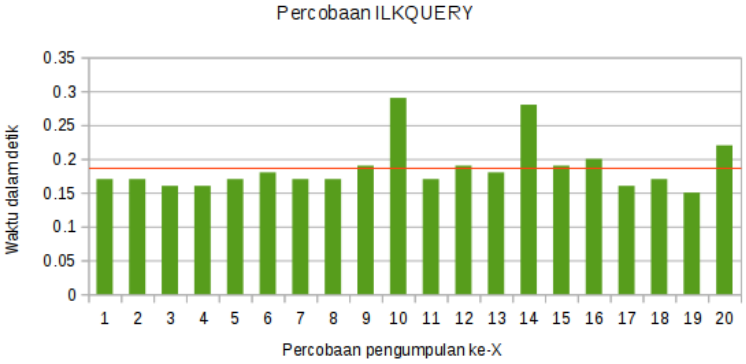
Selanjutnya dilakukan juga uji kebenaran dengan mengumpulkan berkas kode sumber ke situs SPOJ dapat dilihat umpan balik sistem.

Hasil uji kebenaran dan waktu eksekusi program saat pengumpulan kasus uji pada situs SPOJ ditunjukkan pada Gambar 5.8.

18255731	2016-11-25 10:24:23	Fendy	**accepted**	0.16	4.7M	C++14

Gambar 5.8: Hasil uji kebenaran ILKQUERY pada situs SPOJ.

Hal ini membuktikan bahwa implementasi yang dilakukan telah berhasil menyelesaikan permasalahan I Love Kd-Trees dengan batasan-batasan yang telah ditetapkan. Setelah itu dilakukan pengiriman kode sumber implementasi sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Grafik hasil uji coba sebanyak 20 kali ditunjukkan pada Gambar 5.9.



Gambar 5.9: Grafik hasil uji kebenaran ILKQUERY pada situs SPOJ sebanyak 20 kali.

Dari hasil pengumpulan kode sebanyak 20 kali, didapat waktu rata-rata program yaitu 0.187 detik dan penggunaan memori rata-rata yang dibutuhkan program yaitu 4.7MB.

5.2.1.2 Permasalahan I Love Kd-Trees II

Kasus yang akan digunakan sebagai bahan uji kebenaran dalam analisis penyelesaian permasalahan I Love Kd-Trees II menggunakan contoh kasus sebagai berikut:

```
10 3
2 6 7 1 8 1 2 3 2 6
0 0 7 2
1 6
0 0 7 2
```

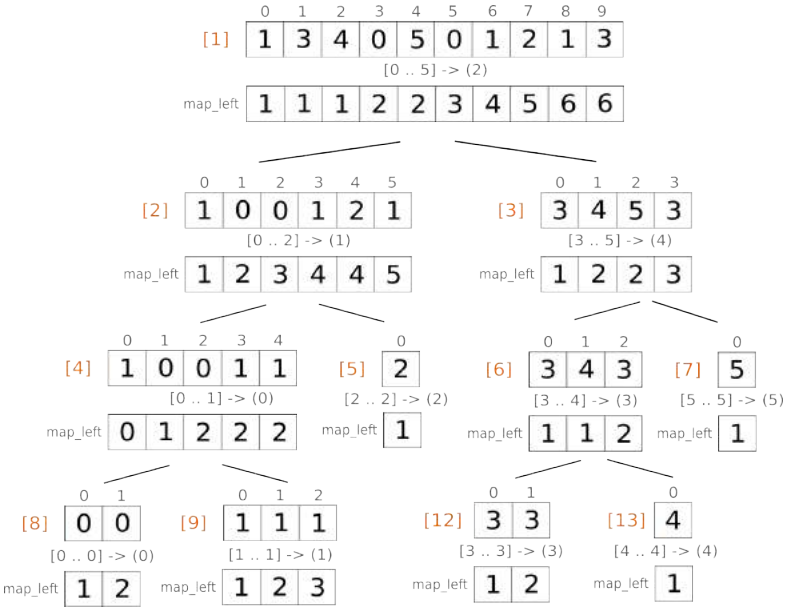
Gambar 5.10: Contoh kasus uji permasalahan I Love Kd-Trees II.

Pengkodean data awal menjadi simbol-simbol yang nantinya digunakan pada permasalahan ini dapat dilihat pada Gambar 5.2.

Sama seperti uji coba kebenaran untuk permasalahan I Love Kd-Trees. Terlebih dahulu perlu dibentuk struktur dari *wavelet tree* untuk membantu penyelesaian *query* yang akan dikerjakan. Dikarenakan data awal pada contoh kasus uji memiliki persamaan maka struktur yang dihasilkan juga sama. Struktur *wavelet tree* setelah pemanggilan fungsi build terlihat pada 5.11.

Berbeda dengan struktur pada Subbab 5.2.1.1 pada *node leaf* struktur *wavelet tree* untuk permasalahan ini juga juga memiliki array map_left. Pada array indeks ke-i menyimpan banyak elemen yang aktif dengan rentang $[1..i]$ yang diimplementasikan menggunakan struktur data BIT. Namun pada Gambar 5.11 tidak ditampilkan struktur internal dari BIT yang ditampilkan adalah representasi dari jumlahan dengan rentang $[1..i]$.

Kemudian *query* pertama pada contoh kasus uji merupakan Operasi Menghitung Jumlah Elemen yang Aktif dengan parameter $i = 0$, $l = 7$, $k = 2$.



Gambar 5.11: Struktur akhir dari *wavelet tree*.

Uji coba untuk *query* pertama adalah sebagai berikut,

Untuk rentang $[0..i-1]$, $id = 1, i = -1, left = 0, right = 5$; pencarian dihentikan karena rentang pencarian tidak valid;

$ans = 0$

Untuk rentang $[0..l]$,

$id = 1, l = 7, k = 1, left = 0, right = 5, mid = 2$; $k \leq mid$; dilanjutkan ke anak kiri;

$id = 2, l = 4, k = 1, left = 0, right = 2, mid = 1$; $k \leq mid$; dilanjutkan ke anak kiri;

$id = 4, l = 3, k = 1, left = 0, right = 1, mid = 0$; $k > mid$;

dilanjutkan ke anak kanan yang merupakan *node leaf*;

ditemukan bahwa indeks pencarian menjadi $l = 3$ namun ternyata pada *node* dengan indeks 9 *node* ini hanya memiliki 3 elemen sehingga nilai kembaliannya adalah 3. Jawaban untuk pertanyaan ini adalah $3 - 0 = 3$.

Query kedua merupakan Operasi Mengubah Status dari Sebuah Elemen dengan parameter $r = 6$. Simbol yang menempati posisi r adalah 1. Sehingga operasi ini akan hanya mempengaruhi *node leaf* untuk simbol 1. Kemudian dilakukan penulusuran untuk mencari posisi *node leaf* yang menyimpan data simbol 1.

$id = 1, r = 6, k = 1, left = 0, right = 5, mid = 2; k \leq mid$;
dilanjutkan ke anak kiri;

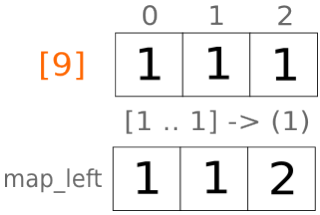
$id = 2, r = 3, k = 1, left = 0, right = 2, mid = 1; k \leq mid$;
dilanjutkan ke anak kiri;

$id = 4, r = 3, k = 1, left = 0, right = 1, mid = 0; k > mid$;
dilanjutkan ke anak kanan;

$id = 9, r = 1, k = 1, left = 1, right = 1, mid = 1$;
node leaf, penulusuran selesai;

Selanjutnya akan dilakukan perubahan pada struktur data BIT pada indeks $r = 1$. Pada indeks ini nilai status akan dikurangkan dengan -1 yang menandakan posisi pada indeks tersebut menjadi non-aktif. Gambar 5.12 menunjukkan struktur dari *node* dengan indeks 9 setelah dilakukan perubahan.

Query ketiga kembali merupakan Operasi Menghitung Jumlah Elemen yang Aktif dengan parameter $i = 0, l = 7, k = 2$. *Query* ini memiliki parameter yang sama dengan *query* pertama. Sehingga tahapan penulusuran sama dengan *query* pertama. Namun dikarenakan telah terjadi proses *update* terhadap status sebuah elemen maka hasil *query* untuk rentang $[0..l]$ menghasilkan nilai yang berbeda.



Gambar 5.12: Struktur *node* dengan indeks 9 setelah dilakukan perubahan.

Seperti terlihat pada Gambar 5.12 nilai BIT untuk $r = 2$ adalah 2 karena terdapat satu elemen yang telah dinonaktifkan.

Pada contoh kasus uji ini hanya *query* pertama dan ketiga yang menghasilkan luaran yaitu, 3 dan 2. Uji coba juga dilakukan dengan mencoba kasus uji pada sistem penyelesaian. Gambar 5.13 menunjukkan bahwa luaran sistem sama dengan hasil uji coba ini.

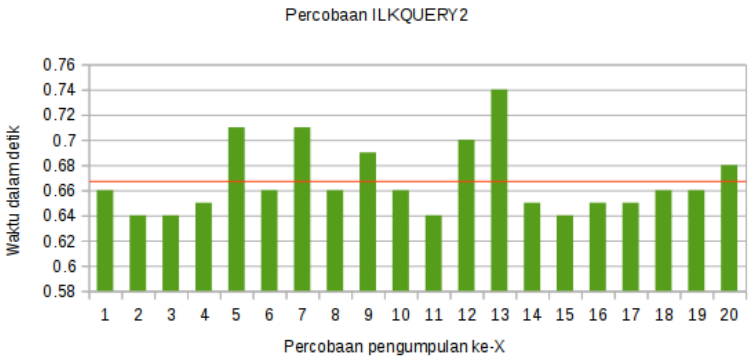
```
10 3
2 6 7 1 8 1 2 3 2 6
0 0 7 2
2
1 6
0 0 7 2
1
```

Gambar 5.13: Hasil luaran program pada contoh kasus uji ILKQUERY2.

Selanjutnya dilakukan juga uji kebenaran dengan mengumpulkan berkas kode sumber ke situs SPOJ dapat dilihat umpan balik sistem. Hasil uji kebenaran dan waktu eksekusi program saat pengumpulan kasus uji pada situs SPOJ ditunjukkan pada Gambar 5.8.

Gambar 5.14: Hasil uji kebenaran ILKQUERY2 pada situs SPOJ.

Hal ini membuktikan bahwa implementasi yang dilakukan telah berhasil menyelesaikan permasalahan I Love Kd-Trees dengan batasan-batasan yang telah ditetapkan. Setelah itu dilakukan pengiriman kode sumber implementasi sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Grafik hasil uji coba sebanyak 20 kali ditunjukkan pada Gambar 5.9.



Gambar 5.15: Grafik hasil uji kebenaran ILKQUERY2 pada situs SPOJ sebanyak 20 kali.

Dari hasil pengumpulan kode sebanyak 20 kali, didapat waktu rata-rata program yaitu 0.6675 detik dan penggunaan memori rata-rata yang dibutuhkan program yaitu 15MB.

5.2.1.3 Permasalahan I Love Kd-Trees III

Kasus yang akan digunakan sebagai bahan uji kebenaran dalam analisis penyelesaian permasalahan I Love Kd-Trees III menggunakan contoh kasus sebagai berikut:

Pengkodean data awal menjadi simbol-simbol yang nantinya digunakan pada permasalahan ini dapat dilihat pada Gambar 5.2.

```
10 3
2 6 7 1 8 1 2 3 2 6
0 2 1 3
1 2
0 2 1 3
```

Gambar 5.16: Contoh kasus uji permasalahan I Love Kd-Trees III.

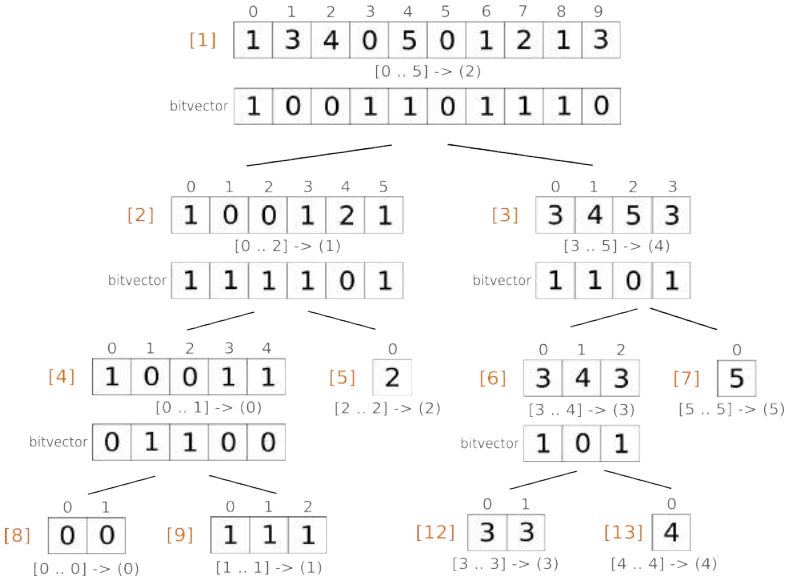
Sama seperti uji coba kebenaran untuk permasalahan I Love Kd-Trees. Terlebih dahulu perlu dibentuk struktur dari *wavelet tree* untuk membantu penyelesaian *query* yang akan dikerjakan. Dikarenakan data awal pada contoh kasus uji memiliki persamaan maka struktur yang dihasilkan juga sama. Struktur *wavelet tree* setelah pemanggilan fungsi build terlihat pada 5.6.

Solusi penyelesaian pada permasalahan ini memerlukan sebuah informasi tambahan untuk mendukung operasi Operasi Menukar Elemen yang Bersebelahan. Tujuannya adalah agar tetap dapat mencapai kompleksitas operasi sebesar $\mathcal{O}(\lg \sigma)$. Informasi tambahan ini akan disimpan dalam array $bitvector$ yang dapat dilihat pada Gambar 5.17. Penjelasan mengenai array telah dipaparkan pada Subbab 3.3.1.

Selanjutnya dilakukan Operasi Mencari Bilangan Terkecil ke-K dengan parameter $i = 2$, $l = 1$, dan $k = 3$. Penulusuran *wavelet tree* dilakukan sebagai berikut,

$id = 1, i = 2, k = 3, left = 0, right = 5, map\_left[id][i] = 1;$
$k > map\_left[id][i];$
dilanjutkan ke anak kanan;

$id = 3, i = 1, k = 2, left = 3, right = 5, map\_left[id][i] = 2;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri;



Gambar 5.17: Struktur *wavelet tree* dengan informasi bitvector.

$id = 6, i = 1, k = 2, left = 3, right = 4, map\_left[id][i] = 1;$
$k > map\_left[id][i];$
dilanjutkan ke anak kanan yang merupakan *node leaf*;

$d = 4$

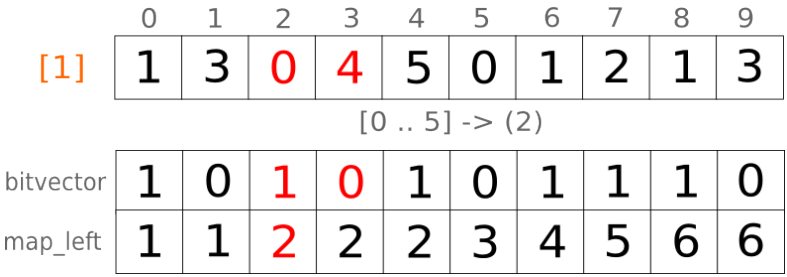
Tabel kemunculan yang digunakan sama seperti pada permasalahan I Love Kd-Trees, dapat dilihat pada Tabel 5.1. Sehingga jawaban untuk *query* ini terdapat pada $occurences[4][1] = 2$.

Query kedua merupakan Operasi Menukar Elemen yang Bersebelahan dengan parameter $i = 2$ maka elemen pada indeks 2 dan 3 akan ditukar posisinya. Hal ini akan menyebabkan struktur dari *wavelet tree* akan berubah. Perubahan dilakukan sesuai algoritma yang telah dijelaskan pada Subbab 2.3.6.

Penulusuran *wavelet tree* untuk mengubah informasi map_left dan $bitvector$ adalah sebagai berikut,

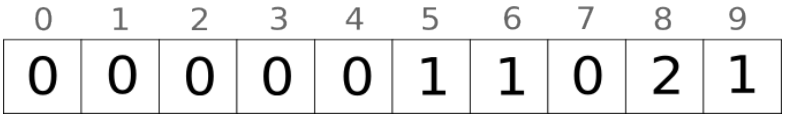
$id = 1$, $i = 2$, $left = 0$, $right = 5$, $bitvector[id][i] = 1$, $bitvector[id][i+1] = 2$; $bitvector[id][i] \neq bitvector[id][i+1]$; dilakukan proses *update* $map\_left[id][i] = map\_left[id][i] + 1$ serta penukaran $bitvector[id][i]$ dan $bitvector[id][i+1]$. Karena kedua elemen telah berada pada subtree yang berbeda maka proses perubahan dihentikan.

Gambar 5.18 menunjukkan kondisi *wavelet tree* setelah dilakukan perubahan. Bagian yang terkena perubahan ditandai dengan warna merah.



Gambar 5.18: Bagian dari struktur *wavelet tree* yang terkena perubahan.

Tabel $occurences$ juga akan mengalami perubahan. Untuk dapat melakukan perubahan pada tabel ini dengan kompleksitas $\mathcal{O}(1)$ maka telah dibangun sebuah array $rev\_occurence$ yang telah dijelaskan pada Subbab 3.3.1. Gambar 5.19 menunjukkan isi dari array $rev\_occurence$.



Gambar 5.19: Isi dari rray $rev\_occurence$.

val \ l	1	2	3
0	2	5	–
1	0	6	8
2	7	–	–
3	1	9	–
4	3	–	–
5	4	–	–

Tabel 5.2: Tabel occurences yang telah diubah.

Perubahan dilakukan ketika simbol pada indeks ke-i tidak sama dengan simbol pada indeks ke-$i+1$. Pada kasus ini simbol yang terpengaruh adalah simbol 4 dan 0. Untuk mengetahui masing-masing simbol merupakan kemunculan yang ke berapa pada array maka digunakan array $rev\_occurence$ yang telah dibangun sebelumnya. Maka $occurences[4][0] = i + 1 = 3$ dan $occurences[0][0] = i = 2$. Tabel yang telah mengalami perubahan ditunjukkan pada 5.2.

Query ketiga memiliki parameter yang sama dengan *query* pertama yaitu, $i = 2$, $l = 1$, dan $k = 3$. *Query* ini bertujuan untuk memaparkan perbedaan yang terjadi karena operasi *update* yang telah dilakukan. Penelusuran *wavelet tree* dipaparkan sebagai berikut,

$id = 1, i = 2, k = 3, left = 0, right = 5, map\_left[id][i] = 2;$
$k > map\_left[id][i];$
dilanjutkan ke anak kanan;

$id = 3, i = 1, k = 1, left = 3, right = 5, map\_left[id][i] = 2;$
$k \leq map\_left[id][i];$
dilanjutkan ke anak kiri;

$id = 6, i = 1, k = 1, left = 3, right = 4, map\_left[id][i] = 1;$

$k \leq map\_left[id][i]$;
dilanjutkan ke anak kanan yang merupakan *node leaf*;

$d = 3$

Sehingga jawaban untuk *query* ini adalah $ans = 1$.

Secara berturut-turut jawaban dari kedua *query* tersebut adalah 2 dan 1. Kemudian sistem penyelesaian dijalankan dan diberi masukan sesuai kasus uji dari analisis sebelumnya dan hasil luaran sistem adalah 2 dan 1 seperti yang terlihat pada Gambar 5.20.

```
10 3
2 6 7 1 8 1 2 3 2 6
0 2 1 3
2
1 2
0 2 1 3
1
```

Gambar 5.20: Hasil luaran program pada contoh kasus uji ILKQUERYIII.

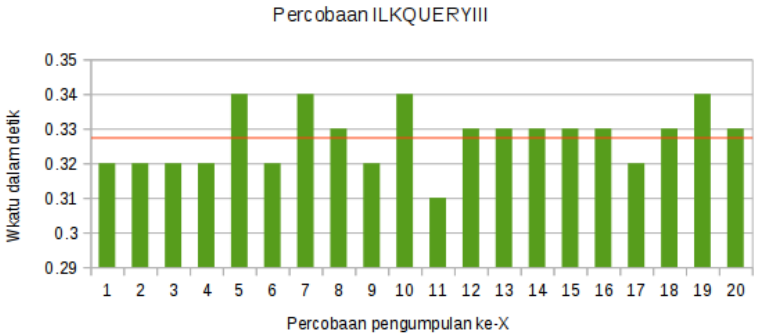
Selanjutnya dilakukan juga uji kebenaran dengan mengumpulkan berkas kode sumber ke situs SPOJ dapat dilihat umpan balik sistem. Hasil uji kebenaran dan waktu eksekusi program saat pengumpulan kasus uji pada situs SPOJ ditunjukkan pada Gambar 5.8.

18298918	■	2016-12-02 08:58:08	I LOVE Kd-TREES III	**accepted** edit ideone it	0.33	8.5M	C++14

Gambar 5.21: Hasil uji kebenaran ILKQUERYIII pada situs SPOJ.

Hal ini membuktikan bahwa implementasi yang dilakukan telah berhasil menyelesaikan permasalahan I Love Kd-Trees dengan batasan-batasan yang telah ditetapkan. Setelah itu dilakukan pengi-

riman kode sumber implementasi sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Grafik hasil uji coba sebanyak 20 kali ditunjukkan pada Gambar 5.9.



Gambar 5.22: Grafik hasil uji kebenaran ILKQUERYIII pada situs SPOJ sebanyak 20 kali.

Dari hasil pengumpulan kode sebanyak 20 kali, didapat waktu rata-rata program yaitu 0.3275 detik dan penggunaan memori rata-rata yang dibutuhkan program yaitu 8.5MB.

5.2.2 Uji Coba Generalisasi

Uji coba generalisasi dilakukan dengan menggunakan permasalahan tambahan dari sistem penilaian daring SPOJ[1]. Permasalahan ini memiliki 2 buah *query* yaitu *query* insert dan *query search*. Untuk *query search* ditanyakan apakah terdapat data pada titik $P(x, y)$ dan pada *query* insert digunakan untuk memasukkan data dengan koordinat (x, y).

Untuk menyelesaikan permasalahan ini dapat dilakukan pemetaan dari setiap titik menjadi sebuah simbol yang unik. Mula-mula status

[1] http://www.spoj.com/problems/ARNAB1

dari setiap simbol adalah tidak aktif. Yang menjadi perhatian disini adalah bagaimana simbol-simbol ini diletakkan tidak akan mempengaruhi hasil *query* dikarenakan rentang pencarian adalah keseluruhan elemen. Dikarenakan sebelumnya telah dilakukan preprocessing titik-titik menjadi simbol-simbol sehingga pertanyaan *query* dapat dipandang sebagai, "Apakah titik(simbol) tersebut berstatus aktif?"

Jika berstatus aktif maka titik tersebut sudah muncul sebelumnya dan sebaliknya. Untuk mengecek kebenaran dari solusi untuk permasalahan ini juga dilakukan pengiriman kode sumber ke situs penilaian daring SPOJ. Gambar 5.23.

Gambar 5.23: Hasil uji kebenaran ARNAB1 pada situs SPOJ.

BAB 6
KESIMPULAN

Pada bab ini akan dijelaskan kesimpulan dari hasil uji coba yang telah dilakukan.

6.1 Kesimpulan

Permasalahan variasi *range query* telah dapat diselesaikan dengan menggunakan struktur data *wavelet tree*. Struktur data ini telah berhasil menyelesaikan variasi *range query* tanpa harus melakukan perubahan yang signifikan pada struktur utama tersebut. Berdasarkan tahapan-tahapan yang telah dilakukan antara lain analisis, desain, implementasi dan uji coba maka didapatkan kesimpulan untuk masing masing permasalahan sebagai berikut,

1. I Love Kd-Trees
   Operasi yang terdapat pada permasalahan ini adalah Operasi Mencari Bilangan Terkecil ke-K. Dikarenakan tidak terdapat operasi *update*, permasalahan ini relatif lebih mudah untuk diselesaikan dan telah dapat diselesaikan dengan cukup efisien. Waktu rata-rata dari 20 pengujian pada situs SPOJ adalah 0.187 detik dan memori yang dibutuhkan sebesar 4.7 MB.
2. I Love Kd-Trees II
   Operasi yang terdapat pada permasalahan ini adalah Operasi Menghitung Jumlah Elemen yang Aktif dan Operasi Mengubah Status dari Sebuah Elemen. Pada permasalahan ini dibutuhkan space yang relatif lebih besar dikarenakan terdapat operasi penghitungan jumlah elemen yang aktif. Waktu rata-rata dari 20 pengujian pada situs SPOJ adalah 0.6675 detik dan memori yang dibutuhkan sebesar 15 MB.

3. I Love Kd-Trees II
   Operasi yang terdapat pada permasalahan ini adalah Operasi Mencari Bilangan Terkecil ke-K dan Operasi Menukar Elemen yang Bersebelahan. Permasalahan ini merupakan pengembangan dari permasalahan I Love Kd-Trees. Waktu rata-rata dari 20 pengujian pada situs SPOJ adalah 0.3275 detik dan memori yang dibutuhkan sebesar 8.5 MB.

Kompleksitas waktu akhir untuk untuk setiap permasalahan adalah $\mathcal{O}(N \lg \sigma + Q \lg \sigma)$ dan kompleksitas memori adalah $\mathcal{O}(N \lg \sigma)$.

DAFTAR PUSTAKA

[1] R. Grossi, A. Gupta, and J. Vitter, **High-order entropy-compressed text indexes**, Proc. 14th SODA, pages 841-850, 2003

[2] H. T. Cormen, E. C. Leiserson, L. R. Rivest and C. Stein, **Introduction to Algorithms Third Edition**, Cambridge, Massachusetts: The MIT Press, 2009

[3] G. Navarro, ***Wavelet* Trees for All**, Journal of Discrete Algorithms, 25:2-20, 2014

[4] R. Castro, N. Lehmann, J. Perez, and B. Subercaseaux, ***Wavelet* Trees for Competitive Programming**, Olympiads in Informatics, Vol. 10, 19-37, 2016

[5] S. Halim, F. Halim, **Competitive Programming 3 : The New Lower Bound of Programming Contests** Singapore: Lulu Publisher, 2013

[6] SPOJ, **ILKQUERY - I Love Kd-Trees**, [*Online*], `http://www.spoj.com/problems/ILKQUERY/`, diakses pada tanggal 22 Desember 2016

[7] SPOJ, **ILKQUERY2 - I Love Kd-Trees II**, [*Online*], `http://www.spoj.com/problems/ILKQUERY2/`, diakses pada tanggal 22 Desember 2016

[8] SPOJ, **ILKQUERYIII - I Love Kd-Trees III**, [*Online*], `http://www.spoj.com/problems/ILKQUERYIII/`, diakses pada tanggal 22 Desember 2016

Halaman ini sengaja dikosongkan

BIODATA PENULIS

Fendy, lahir di Surabaya tanggal 10 Oktober 1995. Penulis merupakan anak kedua dari 2 bersaudara. Penulis telah menempuh pendidikan formal TK Mini Surabaya, SD Katolik St. Theresia II Surabaya (2001-2007), SMP Katolik Mater Dei Probolinggo (2007-2010) dan SMA Katolik St. Louis Surabaya (2010-2013). Penulis melanjutkan studi kuliah program sarjana di Jurusan Teknik Informatika ITS.

Selama kuliah di Teknik Informatika ITS, penulis mengambil bidang minat Dasar dan Terapan Komputasi (DTK). Penulis pernah menjadi asisten dosen dan praktikum untuk mata kuliah Dasar Pemrograman (2014 dan 2015), Struktur data (2015 dan 2016). Selama menempuh perkuliahan penulis juga aktif mengikut kompetisi pemrograman tingkat nasional dan menjadi Juara 1 kategori pemrograman pada lomba Techpouria UNSRI 2016, Juara 2 kategori pemrograman pada lomba Petra Informatics Competition 2016, serta finalis kategori pemrograman pada lomba COMPFEST UI (2015 dan 2016), INC Bina Nusantara (2014, 2015 dan 2016) dan ICPC Regional Asia-Jakarta (2014 dan 2015). Selain itu penulis juga pernah menjadi asisten Pelatnas 2 TOKI (2015). Penulis dapat dihubungi melalui surel di `fendy.fendy95@gmail.com`.